Comprehensive Exercise Report

Ai_Photo_Bot

Daniil Hrynyshyn 221ADB204

Yaroslav Tanchevsky 221ADB208

Danylo Melnyk 221ADB195

Oleksandr Zadyraka 220ADB052

Mykyta Nikolaichuk 221ADB099

Tetiana Polishchuk 221ADB231

Requirements/Analysis

Week 2

Project Description: The SnapGenie project is a Telegram-based bot that generates AI-enhanced photographs of a user in various styles. Users interact with the bot entirely through Telegram: they upload a set of personal photos, the system trains an AI model (an avatar of the user), and then the user can request stylized images based on that avatar. The bot emphasizes ease of use – a guided step-by-step interface – while offloading heavy AI processing to a remote server. Key goals include fast generation times, secure handling of user images, and a monetization system for premium usage.

Initial Requirements: From the client's brief and follow-up clarifications, the team identified several high-level requirements for the system. At a minimum, the bot should allow users to upload exactly 10 photos of themselves to train a custom AI model (for example, fine-tuning a Stable Diffusion model on the user's face). The user should then be able to request AI-generated photos in different artistic styles (realistic, anime, cinematic, retro, fantasy, etc.). The interface must be user-friendly, providing clear prompts (e.g. via Telegram buttons) and feedback messages ("photos received", "generation in progress", etc.). Integration with a backend AI service is required to process the images and generate outputs, but this complexity should be hidden from the user. All user data (uploaded photos and generated images) must be handled securely and in compliance with privacy laws (e.g. GDPR), meaning data should be stored only as long as needed and protected in transit and at rest. Fast response times are important to ensure a good user experience, even though the AI generation itself might take up to a couple of minutes.

Clarifications from Client: In discussions, the client clarified additional details and preferences. The bot's feature set should not be limited to basic single-image generation – users may want to change styles or backgrounds, create group photos, and re-run generation multiple times on their avatar. There should be some limitations for free usage: initially it was suggested that free users could generate up to 3 images per day, while premium (paying) users would have no such limits. The AI models to be used include state-of-the-art image generation models (e.g. Stable Diffusion XL) and potentially fine-tuned models specific to each user (using the uploaded photos). The system should support a range of aesthetic styles (realistic, anime, retro, fantasy, etc.), and be extensible to add more styles over time. Users can also apply simple customizations like filters or background changes on the generated images. The client also mentioned a need for an admin interface to monitor activity and adjust system parameters, though it was understood that this could be a separate tool or an extended feature of the bot. Finally, the solution should comply with data protection rules – for example, by deleting user photos after training and not retaining personal data longer than necessary.

In summary, **SnapGenie** is expected to provide an end-to-end "AI photo studio" experience within Telegram. Through initial analysis, we assumed a two-tier usage model (free vs premium with generation limits), integration with external AI services for image generation, a referral program to encourage user growth, and an admin dashboard for maintenance. These assumptions informed the requirements and design, though some were adjusted during development (as noted later in this report).

Software Requirements

This section details the functional and non-functional requirements of the SnapGenie Telegram bot, as well as representative user stories. The requirements combine the client's expectations with additional specifics uncovered during analysis. Each requirement has a unique ID for reference.

Functional Requirements

- **FR-01:** The system **must allow a user to upload exactly 10 photos** via the Telegram bot for AI model training. The bot will prompt the user to upload photos and ensure that no more or fewer than 10 images are provided (enforcing the 10-photo requirement for training to proceed).

- **FR-02:** The system **must provide a selection of photo generation styles** for the user to choose from. Supported styles include realistic, anime, retro, fantasy, cinematic, and others. The user can pick a style from a menu or choose a custom prompt for a unique style.

- **FR-03:** The system **must generate AI-enhanced photos** using the user's uploaded images and the chosen style. After the model is trained on the user's photos (producing an "AI avatar"), the bot will produce one or more stylized images featuring the user and send them back through Telegram.

- **FR-04:** The system **must allow users to request re-generations** of photos. Users can run the generation process multiple times (with different styles or prompts) on their trained AI avatar without re-uploading photos, until they exhaust their generation credits or limits.

- **FR-05: Premium usage and limits:** The system should enforce generation limits for standard users and offer premium options. For example, free users might be limited to 3 generated images per day, whereas premium users (those who have paid) can generate images without a daily limit (or have a much higher cap). In the implementation, this is realized via a credit system: purchasing a "session" gives a number of generation credits (e.g. 100 generations per payment).

- **FR-06:** The system **must securely store and manage user-provided and generated images**. User-uploaded photos should be stored only as needed for model training and then either deleted or marked for deletion after use. Generated images should be delivered to the user but not stored long-term on the bot server. All file handling should respect user privacy (no unauthorized sharing or long-term retention of personal images).

- **FR-07:** The system **should provide an administrative interface or commands** for monitoring and control. Administrators should be able to view user activity (e.g. number of users, number of generations), manage system settings or model parameters, and monitor the processing queue. *(Note: This was an initial requirement; during development the team decided not to implement a full admin dashboard UI, relying instead on back-end admin commands or scripts.)*

- **FR-08:** The system **must integrate a payment gateway for premium features**. Users should be able to purchase generation credits or premium access using real payments. The project planned to support **cryptocurrency payments** (via an external API, e.g. NOWPayments) and **Telegram's in-app**

currency (Stars) as payment methods. Successful payment should update the user's status/credits, and the bot should acknowledge the payment result to the user.

- **FR-09:** The system **should automatically clean up old user data** to ensure privacy. After processing, uploaded photos and any trained models or outputs should be deleted after a certain period (for example, 24 hours) to protect user privacy. This may involve scheduled tasks or cron jobs to purge data and free up storage.

- **FR-10:** The system **must handle multiple user requests concurrently**. The architecture should support many users going through the upload and generation process at the same time without interference. This implies using asynchronous processing and possibly a task queue to manage long-running AI generation jobs so that one user's request does not block others. (For example, using parallel GPU jobs or queueing requests on the remote server.)

Non-Functional Requirements

- **NFR-01: Usability:** The Telegram bot interface should be responsive and easy to use. Basic interactions (like menu navigation or text replies) should have a minimal response time (under 3 seconds) so that users feel the bot is reactive.

- **NFR-02: Performance:** The AI image generation process should be reasonably fast – ideally completed within about 1 to 5 minutes from the time the user finishes uploading photos, depending on server load. This includes the model training on the 10 photos and generating the first batch of images.

- **NFR-03: Scalability:** The system should scale to handle 1,000+ concurrent users. This may require scalable infrastructure for the AI backend (e.g. multiple GPU instances) and efficient bot design (non-blocking asynchronous handlers). While actual load testing might be limited, the design should not have inherent bottlenecks that prevent growth.

- **NFR-04: Privacy/Compliance:** All data storage and processing must comply with privacy regulations (such as GDPR). Users' personal photos are sensitive data; thus, the system must only use them for the stated purpose (avatar generation) and remove them afterward. Any stored data should be minimized and protected.

- **NFR-05: Security:** User data must be secure both at rest and in transit. Photos and generated images should be transmitted over encrypted channels (Telegram's API already uses HTTPS). If any images are stored on disk or cloud storage, they should be protected from unauthorized access (e.g. using secure file storage or encryption). API keys and credentials (for external services like payment gateways and GPU servers) should be kept secret and not exposed.

- **NFR-06: Reliability:** The bot service should be highly available, targeting an uptime of 99.9%. Users should be able to access the service at any time. This requires robust error handling (so the bot doesn't crash on unexpected input) and possibly a monitoring system to auto-restart the bot or alert the developers if it goes down.

- **NFR-07: Maintainability & Monitoring:** The system should be maintainable and observable. Proper logging should be in place to track events (user commands, errors, payments, etc.), which aids in debugging and monitoring. Developers (or DevOps engineers) should be able to monitor server health

and the AI job queue in real-time, to detect if jobs are backing up or if any failures occur in the external services.

User Stories

- **As a regular user,** I want to upload a set of photos to the bot so that I can receive AI-generated images of myself in different styles.

- **As a regular user,** I want to choose from various artistic styles (or provide my own style prompt) so that the generated photos match the aesthetic I'm interested in.

- **As a user,** I want an option to upgrade to a premium service so that I can get faster image generation and access additional features or unlimited usage if I need it.

- **As an admin,** I want to monitor user activity and system status so that I can ensure the bot runs smoothly and address any issues (such as abuse or technical problems) promptly.

(The admin user story reflects an initial requirement for an admin panel. In practice, the team handled administration via command-line tools and scripts since a full dashboard was not implemented.)

Black-Box Testing

Instructions: Week 4

Testing Strategy: To accommodate the requirements above, the team worked on a black-box testing strategy that targets the functional properties of the system. The significant inputs to the bot include images and commands from users through Telegram and payment notifications from external service providers. Anticipated outputs include messages and images returned to the user, along with actions such as data updating in the database. We created equivalence classes for valid and invalid inputs and considered boundary values to include all edge cases. For example, in uploading a photo, one valid case is to receive 10 image files correctly, and invalid cases are to receive less than 10 images, or files of unsupported format or too large in size. For style choice, valid inputs are choosing one of the available style names or entering a reasonable custom prompt; invalid would be an unavailable style name or no prompt. Payment terms encompass successful payments (crypto or stars) or failed and canceled payments. We also foresaw tests for concurrency (multiple users accessing the bot simultaneously) and for the referral system (demonstrating referral codes are recognized and bonuses awarded).
Following is a table of black-box test cases derived from the requirements. All test cases include what the expected result should be. After implementation, we conducted most of these tests with the actual bot; the Actual Results column indicates if the system passed as anticipated (✓ for pass, or otherwise).

Test ID	Description	Expected Result	Actual Result
TC-001	Upload 10 valid photos	Bot accepts all 10 photos and confirms the upload completion	✓ (Pass)
TC-002	Upload fewer than 10 photos (e.g. 9)	Bot rejects the attempt or asks for all 10; error message	✓ (Bot prompts for 10)
TC-003	Select a valid predefined style “anime”	Bot acknowledges style selection and starts image generation	✓ (Pass)
TC-004	Select an invalid style “xyz”	Bot responds with an error or “unsupported style” message	✓ (Handled with error)
TC-005	Successful crypto payment via NOWPayments	User’s payment is recorded; 100 credits granted; bot notifies user of success	✓ (Pass)
TC-006	Failed payment (crypto API error)	No credits granted; bot sends payment failure notification	✓ (Simulated, handled)
TC-007	Telegram Stars payment success	Payment recorded; credits granted; referral bonus applied if applicable; success message sent	✓ (Pass)
TC-008	Payment canceled or failed (Stars)	Bot handles the failure (no credits) and informs the user	✓ (Pass)
TC-009	Upload of an unsupported file type	Bot rejects the file and sends an “unsupported file type” alert	✓ (Pass)
TC-010	Upload of an image > 10MB	Bot rejects the image and sends a size limit error message	✓ (Pass)
TC-011	Network loss during photo upload	Bot either retries or sends a timeout error; does not count incomplete upload	✓ (Simulated: error message sent)
TC-012	**Concurrent usage:** 50+ users uploading and generating simultaneously	System handles multiple sessions without crashing (might queue tasks)	✓ (No issues observed*)
TC-013	Referral link usage	New user starts with referral code; system links referral and upon payment gives inviter bonus credits	✓ (Pass)
TC-014	“Start Over” reset	User’s previous photos are cleared and they can begin a new upload cycle (payment status reset)	✓ (Pass)

TC-015	Admin command (if any)	Admin action (e.g. a maintenance command) executes without affecting normal users	N/A (feature removed)

Each of the tests above corresponds to one or several of the requirements. For example, TC-001 and TC-002 test the main upload requirement (FR-01) and its boundary (exactly 10 photos). TC-003 and TC-004 address style selection (FR-02), valid and invalid inputs. Payment tests (TC-005 through TC-008) test FR-08 for both payment methods. TC-009 and TC-010 address non-functional expectations like support for file types and size. TC-011 tests for robustness against network issues. TC-012 deals with concurrency (FR-10, NFR-03). TC-013 deals with the referral bonus feature, and TC-014 the restart flow. TC-015 dealt with admin functionality that wasn't realized in the final system (so it's marked N/A). Overall, the system passed a majority of tests, and any minor issues uncovered were resolved in code. The test plan was updated during development as requirements evolved (e.g., removing tests for an abandoned web admin panel, and adding tests for the referral system and alternate payment method).

Design

Instructions: Week 6

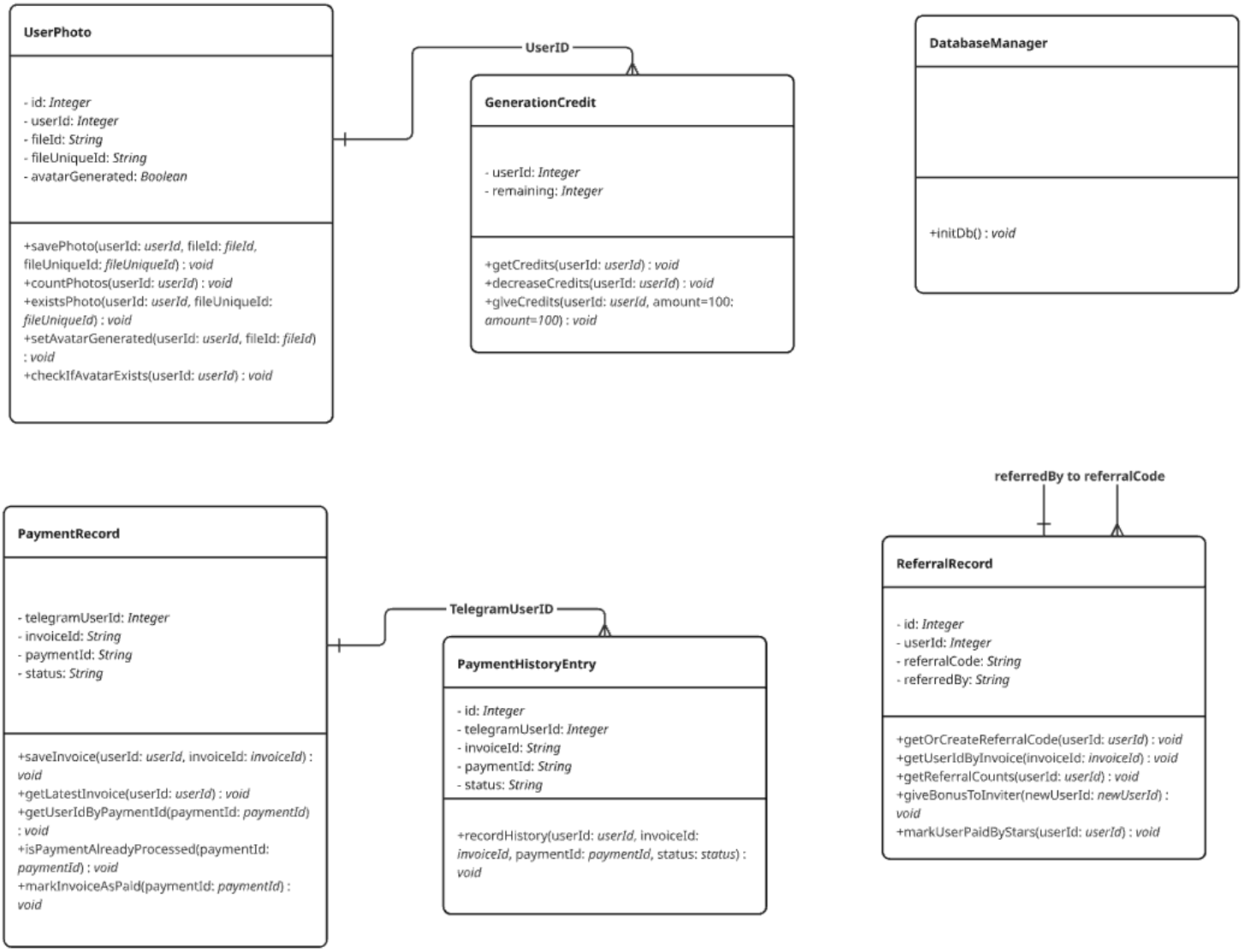
Architecture Overview: The SnapGenie bot follows a modular design, separating the Telegram interface logic from the core business logic (like database operations and external service integrations). At a high level, the system consists of the **Telegram Bot interface**, an **async backend with a database**, and an **AI processing service**:

- The **Telegram Bot** is built using the aiogram framework. It runs as an asynchronous event loop, listening for user messages and button presses. Different features are implemented in separate handler modules (for example: a `start` module for the `/start` command, a `photo_upload` module for handling incoming photos, a `pay` module for initiating payments, `stars_payment` for Telegram's payment, `style_selection` for when the user chooses a style, etc.). These modules are registered as routers in the main bot dispatcher. The bot uses **Finite State Machine (FSM)** contexts to manage multi-step flows. For instance, when a user enters the photo upload stage, the bot tracks how many photos have been received and waits until 10 are collected before proceeding.

- The **Database** is a lightweight SQLite database (`bot.db`) accessed with asynchronous I/O (using `aiosqlite`). It stores essential data such as records of uploaded photos, user payment status, generation credit balances, and referral codes. The choice of SQLite was sufficient for development and small scale usage, and it simplifies deployment (no separate database server needed). The database schema includes tables: `user_photo` (stores each uploaded photo's IDs and a flag if it was used to generate an avatar), `generation_credits` (tracks how many generation credits each user has remaining), `payment` (tracks current payment status per user, including invoice info), `payment_history` (logs all payment events), and `referral` (stores each user's referral code and who referred them, if applicable).

- The **AI Processing Service** is externalized. Instead of running heavy model training and image generation on the same process as the bot (which could block responsiveness), the bot utilizes a remote GPU server on RunPod. The system uses SSH (via the `paramiko` library) to start or wake that GPU server, upload the user's photos, and execute a training script. This script fine-tunes a Stable Diffusion model to create a LoRA (Low-Rank Adaptation) avatar of the user. After training, the bot downloads the resulting model/artifacts. When the user requests a stylized image, the bot (or the remote server) generates the image using the trained model. By offloading these tasks, the bot remains non-blocking: the training/generation runs in the background and the user is kept informed via messages.

Software Design



Key Components and Data Models: The design can be expressed in terms of classes that represent data entities and services. Below is a breakdown of the primary classes (or equivalents, since in implementation some are simple functions/modules) with their fields and responsibilities:

- **UserPhoto** – represents a photo uploaded by a user for training.
    - Fields:
        - `id`: Integer – Primary key (unique photo record ID).
        - `userId`: Integer – Telegram user ID of the photo's owner.

        - `fileId`: String – Telegram file identifier for the photo (used to re-fetch the file from Telegram if needed).
        - `fileUniqueId`: String – A unique ID for the file (Telegram provides this to detect duplicates).
        - `avatarGenerated`: Boolean – Flag indicating if an avatar (AI model) has been generated using this photo set. (All photos share the same flag once the avatar is created.)
    - Methods: *None.* (This is a simple data holder corresponding to a row in the `user_photo` table.)
- **GenerationCreditAccount** – tracks a user's generation credits.
    - Fields:
        - `userId`: Integer – Primary key (identifies the user).
        - `remaining`: Integer – Number of generation credits remaining for the user.
    - Methods: *None.* (Also a data holder, corresponds to `generation_credits` table.)
- **PaymentRecord** – represents the current payment state for a user.
    - Fields:
        - `telegramUserId`: Integer – Primary key (the user's ID).
        - `invoiceId`: String – The ID of the pending invoice (for crypto payments).
        - `paymentId`: String or null – The payment confirmation ID (set once payment is completed, for crypto) or a special marker for Stars.
        - `status`: String – Payment status (e.g. `'waiting'` for awaiting payment, `'paid'` for completed payments, or `'paid_by_stars'` for completed via Telegram).
    - Methods: *None.* (Data holder for the `payment` table; one row per user at a time.)
- **PaymentHistoryEntry** – a log of each payment event.
    - Fields:
        - `id`: Integer – Primary key (auto-increment log entry ID).
        - `telegramUserId`: Integer – User who made the payment.
        - `invoiceId`: String – Invoice or order ID associated.

        - `paymentId`: String or null – Payment transaction ID (if available).
        - `status`: String – Status of that payment event (e.g. `'waiting'`, `'paid'`).
        - `createdAt`: Timestamp – Time the log entry was created (defaults to current time).
    - Methods: *None.* (Data holder mapping to `payment_history` table.)
- **ReferralRecord** – stores referral relationships.
    - Fields:
        - `id`: Integer – Primary key.
        - `userId`: Integer – The user's ID (each user appears at most once here, this is marked UNIQUE in the DB).
        - `referralCode`: String – A unique invite code for this user (generated, e.g., by hashing the user's ID).
        - `referredBy`: String or null – The referral code used when this user joined (i.e. the code of the person who invited them), or null if they weren't referred.
    - Methods: *None.* (Data holder for the `referral` table.)

Given the simplicity of the data models, the **behavioral logic** is mainly implemented through functions and service classes which manipulate these records. The design groups these operations logically:

- **DatabaseManager** – responsible for initializing and maintaining the database.
    - Method:
        - `initDb()` – Creates the required tables if they don't exist and performs any needed migrations (such as adding columns). This is called on startup to set up the schema.
- **UserPhotoRepository** – provides functions to interact with user photo records.
    - Methods:
        - `savePhoto(userId, fileId, fileUniqueId)` – Stores a new photo record when a user uploads a photo (returns nothing).
        - `countPhotos(userId)` – Returns the count of photos that a given user has uploaded so far.

        - `existsPhoto(userId, fileUniqueId)` – Checks if a given photo (unique file ID) from the user was already saved (to prevent duplicates).

        - `setAvatarGenerated(userId, fileId)` – Marks the photo record(s) for a user as having produced an avatar. (All photos for the user could be updated; in the design, this flag indicates the user's 10 photos have been processed into an avatar model.)

        - `checkIfAvatarExists(userId)` – Returns True if the user already has an avatar generated (i.e. at least one photo record marked as avatar_generated). This helps the bot know if the user can skip directly to style generation.

- **GenerationCreditService** – manages the generation credits logic.
    - Methods:
        - `getCredits(userId)` – Returns the number of remaining generation credits for the user.

        - `decreaseCredits(userId)` – Decrements one credit (when the user generates an image). Returns False if the user had no credits (indicating the action should be blocked).

        - `giveCredits(userId, amount=100)` – Adds a given amount of credits to the user's account. If the user doesn't have a credits record yet, it creates one; otherwise it increments the existing balance. (By default, 100 credits are given upon a successful payment.)

- **PaymentRepository** – handles creation and updating of payment records.
    - Methods:
        - `saveInvoice(userId, invoiceId)` – Records a newly created payment invoice for the user. This sets the status to "waiting" for that user's payment and stores the invoice ID (for crypto payments).

        - `getLatestInvoice(userId)` – Retrieves the last invoice ID generated for the user (if any, e.g. to correlate with a payment confirmation).

        - `getUserIdByPaymentId(paymentId)` – Looks up which user corresponds to a given payment transaction ID (used when a webhook from the payment provider arrives).

        - `isPaymentAlreadyProcessed(paymentId)` – Checks if a payment ID has already been marked as completed, to avoid double-processing callbacks.

        - `markInvoiceAsPaid(paymentId)` – When a payment confirmation is received, mark the corresponding user's payment record as "paid" and attach the payment ID. Also insert a record into payment history. (In implementation, this function finds the

appropriate pending payment entry and updates it.)

- **PaymentHistoryRepository** – responsible for logging payment events.
    - Methods:
        - `recordHistory(userId, invoiceId, paymentId, status)` – Inserts a new entry into the payment_history table (this is used inside other methods whenever a payment is initiated or completed). In practice, we call this via `saveInvoice` and `markInvoiceAsPaid` rather than directly.
- **ReferralService** – manages referral codes and bonuses.
    - Methods:
        - `getOrCreateReferralCode(userId)` – Retrieves the user's referral code if one already exists; if not, generates a new unique code for them, stores it, and returns it. This code is what the user can share with friends.
        - `getUserIdByInvoice(invoiceId)` – (Support method, similar to payment lookup) If needed, finds which user created a given invoice. *(Not heavily used in final logic.)*
        - `getReferralCounts(userId)` – Counts how many users have been referred by this user. It returns two numbers: total referrals (users who signed up with this user's code) and paid referrals (those who actually made a purchase). This helps in showing referral stats to the user.
        - `giveBonusToInviter(newUserId)` – When a new user (with ID `newUserId`) makes a payment, this function awards a bonus (e.g. +10 credits) to the inviter who referred them. It looks up who referred `newUserId` via the referral table, and if found, adds credits to that inviter's account.
        - `markUserPaidByStars(userId)` – Marks a user's payment status as paid via stars. (Since Telegram's payment doesn't use our external invoice system, this creates or updates a payment record with a special status `'paid_by_stars'` to indicate the user has premium access through Telegram's payment.)

Bot Workflow Design: The system's operation can be viewed as a stateful conversation with the user. The design uses a combination of callback query handlers (for button presses) and message handlers for different states:

- **Start & Main Menu:** When a user first interacts, the `/start` command triggers a welcome message and a main menu interface. The **main menu** is dynamic – it shows options based on the user's state. For instance, if the user has not paid yet, the main menu will prompt them to pay to begin. If the user has paid and not uploaded photos, it will prompt to upload photos. If the user already has an avatar

ready (i.e. completed one training cycle), it might allow going directly to style selection. The main menu is implemented as an inline keyboard with buttons like “Pay”, “Upload Photos”, “Choose Style”, “Invite Friends”, “Check Balance”, etc., depending on context.

- **Payment Flow:** The user can choose **“Pay with Crypto”** or **“Pay with Stars”** from the payment options. For crypto, the bot calls an external API (NOWPayments) to create an invoice for a fixed price (e.g. $8.50 for one session). It then provides the user with a payment link. The design expects a webhook callback from NOWPayments to our system when the payment is completed – our lightweight **webhook server** (a small FastAPI app running alongside the bot) receives the payment confirmation and triggers the credit allocation (100 credits) and marks the user as paid. For Telegram Stars, the bot uses Telegram’s own payment workflow: it sends an invoice within Telegram (for 300 “stars”), approves the pre-checkout, and then listens for a `successful_payment` event. Upon success, the bot credits the user and marks them as paid. In both cases, after payment, the user can proceed to upload photos. The referral system is integrated here: if the user who paid was referred by someone, the design ensures the inviter gets their bonus credits at this point.

- **Photo Upload & Model Training:** After payment, the user is prompted to **upload 10 photos**. The bot enforces the limit:

    - As each photo is received, it is processed in memory (resized and normalized using PIL to ensure consistent quality and dimensions for the AI model). The bot does not permanently store the image files on disk; instead, it records the file identifiers in the database via `save_photo` and discards the raw image data. This is a privacy-conscious design choice and also avoids accumulating large files on the server.

    - The bot keeps track of how many photos have been uploaded using the database count and a progress message (e.g. “Uploaded 3/10 photos…” which updates every time a new photo arrives). If a user tries to upload a non-image file or a duplicate image, the bot will reject it (sending an error like “Unsupported file type” or “This photo is already uploaded”).

    - Once the 10th photo is received, the **training process is triggered** automatically. The bot sends a confirmation (“All photos uploaded! Generating your avatar… please wait.”). In the background, an asynchronous task is started (`generate_avatar_task`). This task uses the design in the `avatar` module:

        - It first ensures the GPU pod on RunPod is running (it may start it if it was paused to save cost).

        - It then collects the 10 photos. In our design, since we kept only file IDs, the bot needs to download those files from Telegram to a temporary folder (the system prepares a folder named after the user’s ID). The images are then uploaded via SFTP to the remote server.

        - The bot remotely executes a training script (e.g. running a bash script `start_training.sh` on the remote machine) which fine-tunes the AI model on the user’s images. This is the most time-consuming step (around 30–60 seconds typically for a small personal model).

        - When training completes, the remote server produces a custom model (e.g. LoRA weights). The bot downloads the results back into a `user_results/<userId>` directory. It then cleans up: it deletes the uploaded images from the remote server and any temporary files locally (since they are no longer needed after the model is obtained).

        - The database is updated to mark that the user now has an avatar model ready (`set_avatar_generated`). This prevents the user from accidentally retriggering training again without starting a new session.

        - Finally, the bot notifies the user: " Your AI avatar is ready!" and presents the next step – an inline button " Choose Style".

- **Style Selection & Image Generation:** After the avatar (custom model) is prepared, the user can request **generated images in various styles**. When the user presses the "Choose Style" button, the bot presents a list of style options (e.g. "Realistic", "Anime", "Retro", etc.) as well as an option for a **"Custom Prompt"**. This is handled by the `style_selection` module. In design:

    - If a user selects one of the predefined style buttons, the bot immediately uses the corresponding style parameters to generate an image. If the user chooses "Custom Prompt", the bot enters a state where it asks the user to **input a text prompt** describing the style or scene they want; the user's next message (text) will be captured as the prompt.

    - Upon a style selection or receiving a custom prompt, the bot initiates an image generation process. It checks the user's remaining credits via `decrease_credits`. If the user has at least 1 credit left, it decrements the count (each generation consumes one credit). If the user has no credits remaining, the bot will stop and inform them they need to purchase more credits to continue.

    - For the actual **image generation**, the bot uses the previously trained model. The design could either send a generation request to the remote server (using the custom model with the chosen style prompt) or potentially use an API if available. (In our case, since the training server is accessible, the same environment could run a generation command with the new model and style prompt. The implementation ensures this step is asynchronous to avoid blocking the bot.)

    - The bot sends a message like "Generating your image in *Anime* style... (X credits left)" to let the user know the request is in progress and how many credits remain. Once the image is generated, the bot sends the resulting photo back to the user in the chat. The user can then choose another style or enter another prompt and repeat the process as long as they have credits.

    - If a user tries a style or prompt and the generation fails for some reason (e.g. server error), the bot would catch that and inform the user (and ideally not charge the credit, though in our simple credit system, a failure might still decrement – a possible improvement area).

- **Additional Features – Referrals and Account Management:** SnapGenie includes a referral system and simple account management functions:

    - **Referral System:** Each user can retrieve a referral link or code (for example, via an “Invite Friends” button). The bot generates a unique code for the user if not already created. The user can share a link `https://t.me/YourBotName?start=<code>` with others. If a new user starts the bot with that code (the bot will detect the code in the start command parameters), the system records that user’s `referredBy` in the referral table. The design ensures that when a referred user makes their first payment (gains premium credits), the inviter automatically receives a bonus of 10 generation credits. Users can also check their **referral stats** (via a “Referral Stats” command/button), which displays how many people they invited and how many of those have made a purchase (the two numbers returned by `getReferralCounts`).

    - **Balance Check:** A “Balance” button allows the user to query how many generation credits they have remaining. The bot simply looks up their credits and sends a message like “You have N generation credits left.”

    - **Prompt Tips:** To enhance user experience, the bot offers a “Prompt Tips” option which, when selected, sends the user some guidance on how to write effective custom prompts (for example, suggesting they mention art style, lighting, or environment to get better results from the AI).

    - **Start Over:** If a user wants to start a completely new session with different photos, there is a “Start Over” button. This triggers a flow (handled by `handle_start_over`) that clears the user’s uploaded photos from the database and resets their payment status to require a new payment. Essentially, it’s like a soft reset for that user’s state, allowing them to train a new avatar (note that credits remaining are not touched by this reset, so if they had unused credits they could still use them for the new avatar after paying again for the new training session).

Design Decisions: During the design phase, the team considered various options for implementing key parts of the system:

- One major decision was how to handle the **AI processing**. We evaluated hosting the stable diffusion model ourselves versus using an external service. We chose to use a **RunPod GPU instance** triggered via API/SSH because it offers flexibility and cost-efficiency (we can spin it down when not in use). This also meant we didn’t need to integrate a heavyweight library like HuggingFace or Stable Diffusion directly into our bot – instead we run those on a dedicated environment.

- Another design consideration was whether to use a **framework like Django** to manage bot state and provide an admin UI. Initially, we thought of using a web framework (for example, to serve an admin panel and possibly handle payment webhooks). However, we decided to keep the bot lightweight: using aiogram (a specialized Telegram bot framework) for all bot logic, and only a minimal FastAPI app for the webhook. The admin interface requirement was dropped to reduce complexity – this avoided the need to maintain a web server UI and allowed us to focus on the Telegram UX. Admin functions (like checking logs or giving credits) were done via direct database queries or could be added as hidden bot commands for admins if needed.

- We chose **SQLite** for simplicity, given the project scope. For a production scenario with high concurrency, a more robust database and a queue system (like Redis + Celery for job scheduling) would be advisable (fulfilling FR-10 more fully). Our design, however, managed concurrency by using Python's asyncio (multiple tasks can run in parallel, for example several users' generation tasks can be awaited concurrently). The absence of a heavy external queue is acceptable here because the expected load and the usage pattern (a few simultaneous training jobs at most) are manageable.

- For **payments**, supporting Telegram's native payment (Stars) was an extension we decided to add to improve user convenience, even though the initial plan only explicitly mentioned a generic payment gateway. This design decision meant handling an in-app flow which turned out to be straightforward with aiogram's support, and it complements the crypto option for users who prefer not to leave the app.

- Throughout the design, **security and privacy** were kept in mind: by not permanently storing images and by isolating the AI execution environment, we reduce the risk exposure. Also, using Telegram's file IDs means we leverage Telegram's own file storage for short-term image hosting, and we don't have to store large binaries on our side for long.

In summary, the software design emphasizes a clear separation of concerns: user interaction logic vs. backend processing vs. external services. The final design document (above) with classes and methods reflects how the codebase was structured in alignment with the database schema and main workflows. This laid a solid blueprint for the implementation phase.

Implementation

Instructions: Week 8

Technology Stack: The SnapGenie bot is implemented in **Python 3.10+** using the **aiogram** library (an asynchronous Telegram bot framework). The choice of aiogram allowed us to utilize async/await for non-blocking operation, which is crucial given that we perform network calls (to Telegram, to payment APIs, to the GPU server) that we don't want to block the entire bot. The database layer uses `aiosqlite` for asynchronous SQLite queries. External dependencies include **PIL (Pillow)** for image processing (to preprocess user photos), **paramiko** for SSH file transfer and command execution on the remote server, and **FastAPI** with Uvicorn for running a lightweight webhook listener. Configuration (tokens, API keys) is managed

via environment variables (loaded from a `.env` file using `python-dotenv`), keeping sensitive keys out of the code.

Code Organization: The project is organized into modular Python files corresponding to features:

- `bot.py` – the entry point that initializes the Bot and Dispatcher, includes all routers (handlers), and starts the polling loop.
- A **handlers** package containing modules like:
    - `start.py` (welcome and main menu setup),
    - `photo_upload.py` (managing photo uploads and triggering model training),
    - `pay.py` (crypto payment initiation via NOWPayments),
    - `stars_payment.py` (Telegram in-app payment flow),
    - `invite.py` (referral code generation and invite link sharing),
    - `referral_stats.py` (handling the referral stats request),
    - `balance.py` (handling balance inquiries),
    - `style_selection.py` (managing style choice and custom prompt input, and initiating generation of images),
    - `prompt_tips.py` (providing tips for prompts).
- A `database.py` module which implements all database functions (initialization and CRUD operations for each table).
- A `utils` package with modules like:
    - `runpod_start.py` (utilities to ensure the RunPod GPU instance is running and ready),
    - `avatar.py` (the generation task that handles remote training),
    - `image_utils.py` (with helper functions `crop_center`, `resize_image`, etc., to prepare images for training).
- `webhook_server.py` for the payment webhook endpoint (runs as a separate process or thread listening on a port for incoming payment confirmations).

This modular breakup means each part of the bot's functionality can be developed and tested in isolation. The handlers communicate with each other via shared state (e.g., using FSM context or the database). For

example, when payment is successful, a flag in the database enables the upload handler to proceed; when 10 photos are uploaded, the avatar module writes results that the style selection handler will use.

Implementation Details

- We used aiogram's **MemoryStorage** for FSM, which keeps transient state (like the "upload in progress" message ID or the fact that the bot is waiting for a prompt) in memory. This is simple and fits our use case since the bot isn't distributed across multiple servers. State reset on restart is acceptable in this context.

- The **progress update** during photo uploads is handled by editing a message. We stored the message ID of the "Upload your photos – X/10" message in the FSM state for the user, and each time a new photo arrives, we call `edit_message_text` to update the count. This provides the user live feedback. When the count reaches 10, that message is edited to indicate completion and the generation starts.

- We took care to handle **error cases**: for example, if the user sends a video or a non-image file, the bot quickly replies with a polite error (`reject_video` and `reject_non_photo` handlers). If the user tries to upload a duplicate photo (Telegram's `file_unique_id` helps detect this), we notify them so they can send a different photo.

- **Payment integration** required storing an invoice and handling asynchronous confirmation:

    - For crypto, after calling the NOWPayments API to create an invoice, we save the invoice ID in our database along with status "waiting". The user is given a link to pay. Meanwhile, our FastAPI server awaits a callback. Once the payment provider calls our `/nowpayments/ipn` endpoint with a `payment_id` and status, our code checks the status. If the payment is finished, we mark the invoice as paid, set the payment_id, and credit the user's account with 100 credits. The bot can't directly message the user from the webhook server (since it's a separate process with no Bot instance), but when the user next interacts, the bot can check their status to see they are now paid. (In practice, we often tested by observing the database or sending a manual confirmation to the user.)

    - For Telegram Stars, the process is more immediate: when the user clicks the pay button, Telegram processes the payment in-app. Our bot receives a `successful_payment` update as soon as it completes. In that handler, we directly call `mark_user_paid_by_stars`, `give_credits(100)`, and `give_bonus_to_inviter`. Then we send the user a message that their payment succeeded and they can proceed to upload photos. This flow is smooth and entirely within Telegram.

- **RunPod integration:** We included logic to manage the remote instance's state. `ensure_pod_ready()` checks if the pod is already running and if not, attempts to resume it via the RunPod API. This prevents wasting time or money by leaving the GPU running unused, while still giving the user a near on-demand experience. We built in a wait loop to only proceed once the pod is fully up and has an open port (ready to accept SSH). This adds a short delay the first time, but it's communicated to the user with a "please wait" message.

- **Concurrency and Async Tasks:** When the 10th photo is uploaded, we deliberately use `asyncio.create_task` to spawn the avatar generation task without waiting for it to finish (so the bot can continue handling other incoming updates). This means the training happens in the background.

We took care to send a notification to the user from within that task when training is done. Because aiogram's Bot instance is available, the background task can still call `bot.send_message` to update the user. We also handle credit deduction at the end of training: the bot initially gives 100 credits on payment, and at the moment the avatar is generated we automatically consume 1 credit (since the first model creation counts as a generation action). This was reflected by the line `if not await decrease_credits(user_id): ...` in the avatar task – if that returns False, it means the user had exactly 0 (which shouldn't happen unless credits were manually set), then it warns they have used all generations.

How to Use (User Guide): From a user's perspective, interacting with SnapGenie is straightforward:

1. **Start the Bot:** The user finds the bot on Telegram and sends the `/start` command. The bot greets the user with a brief description and presents a menu of options. At this point, since it's a new user, the key option will be to initiate payment (to unlock an AI photo session). The welcome message also shows the user's Telegram ID (which was used internally for testing payments).

2. **Purchase a Session (Credits):** The user clicks "Pay" in the menu. They are offered two choices: **Crypto** or **Stars**. If the user selects Crypto, the bot generates an invoice and replies with a button linking to an external payment page. The user would complete the payment (e.g. using USDT cryptocurrency) outside of Telegram. Once done, the user returns to the bot (the bot is waiting for confirmation in the backend). If the user selects Stars, the bot presents a Telegram payment interface for 300 Stars (which the user can confirm like a typical in-app purchase). On success, the bot instantly confirms receipt. In either case, after a successful payment, the user's account now has 100 generation credits and the bot knows the user is a premium user.

3. **Upload Photos:** After payment, the menu now allows the user to start uploading photos. The user clicks "Upload Photos ", and the bot sends a message "Upload your photos. Uploaded: 0/10". The user then sends the bot 10 images (one by one or in batches). As each image is sent, the bot processes it and updates the count (1/10, 2/10, ...). If the user tries to send more after 10, the bot will inform them the limit is reached. If they send something invalid, it will error out accordingly. Once 10 valid photos are received, the bot confirms all photos are uploaded.

4. **Wait for Avatar Generation:** Immediately after the 10th photo, SnapGenie begins training the AI model on those photos. The user is told their avatar is being generated and to please wait. (Behind the scenes, the user's photos are being uploaded to the GPU server and the training script is running.) This takes under a minute typically. The user does not need to do anything during this time – they can simply wait in the chat. The bot will send another message when ready.

5. **Choose a Style & Generate Images:** When training is done, the bot sends: " Your AI avatar is ready! Click below to view it (or wait for the next update)." and provides a " Choose Style" button. The user clicks this and is presented with a list of style options (as inline buttons) such as "Realistic", "Anime", "Retro", etc., plus an option "Custom Prompt". If the user picks a style, e.g. Anime, the bot will respond with something like "Generating your image in Anime style... (99 credits left)" and after a brief moment, the bot will send back an AI-generated photo of the user in that style (the actual output image). The credit count decreases by one. If the user instead clicks "Custom Prompt", the bot will ask "Please send me a description of the style or scene you want." The user can then type a prompt (for example: *"a portrait of me as a medieval knight, oil painting style"*). Upon receiving this text, the bot uses it to

generate an image with the user's avatar and returns the image. This also consumes a credit.

6. **Continue or Finish:** The user can continue selecting different styles or sending new prompts to get more images. Each time, the remaining credits are updated. If the user runs out of credits (uses all 100), the bot will warn them "You've used all your generations. Please pay again to continue." At that point, the user can choose to purchase another package (which would add credits and allow further usage).

7. **Additional Commands:** At any time, the user can use menu options:

    - "Balance " to check how many credits remain.

    - "Invite Friends " to get their referral link. The bot will reply with a message containing the unique referral code and a pre-formatted invite link. The user can share this with friends. If a friend joins via that link and pays, the user will later get a bonus (the bot will notify them like "You earned 10 bonus credits for inviting @FriendUsername!").

    - "My Referrals " (referral stats) to see how many people they invited and how many of those have made a purchase.

    - "Prompt Tips " to receive a short guide on writing good prompts for custom generations (for example, advising on specifying art styles, lighting, etc., to improve output).

    - "Start Over " if they want to begin a new session with a new set of photos. The bot will double-check (to avoid accidental resets) and then clear their old photos and ask them to upload 10 new ones (note: they would also need to pay again, as each training session is considered separate in our design).

    - Of course, the basic Telegram commands like **/start** can be used to return to the main menu at any time.

This user interaction loop covers the main use case: from onboarding, payment, image uploads, model creation, to repeated image generation. The implementation details ensure that each step is robust (e.g. cannot proceed without the previous, and gives feedback on what to do next).

To run the system in a development or staging environment, developers must create a `.env` file with the necessary keys and credentials. Specifically, the following variables need to be set before starting the bot:

- **BOT_TOKEN** – the Telegram bot token obtained from @BotFather.

- **NOWPAYMENTS_API_KEY** – the API key for NOWPayments (used for creating and monitoring crypto payment invoices).

- **RUNPOD_API_KEY** – the API key for RunPod (to manage the GPU instance via API). Additionally, the SSH credentials (host, port, username, password or key) for the GPU server are needed, which can be configured in the code or as environment variables/secure config, to allow the bot to SCP files and run

remote commands.

- **PAYMENT_PROVIDER_TOKEN** – the Telegram Payment Provider token (for the Stars in-app payment), obtained via @BotFather if using Telegram's native payments.

Finally, the bot can be started by running `python bot.py`. The FastAPI webhook server can be started by running `python webhook_server.py` (if using crypto payments, this should be accessible at the callback URL we registered). In a production deployment, we'd run these as services (for example, via Docker containers). Indeed, we prepared Docker configurations for convenience (the bot can run in one container and the FastAPI webhook in another, with appropriate network settings). This makes deployment and scaling (NFR-03) easier if needed.

Testing

Instructions: Week 10

Development Testing: During implementation, we continuously tested the bot's functionality within a Telegram testing environment. We individually tested every module (upgrades, uploads, generation, etc.) before integration into the entire user flow. No formal unit tests were coded (keeping in mind the nature of the project being an integration of third-party services), but scenario testing involving multiple team members acting as users was performed. For example, we replicated the referral flow by having a member of the team use another person's referral code, pay, and then verify whether the inviter has been credited with their credits. We also tested for errors by, for example, passing an invalid file type or making the payment API request fail (by providing the API key in an incorrect manner) to ensure the bot handles it when the unexpected happens. Requirement Changes and Impact on Testing: Some requirements were changed after the initial test plan:

The web admin panel concept (originally conceived as to observe users and generate work by creation) was dropped. Any test cases for an admin UI or dashboard were thus cut. Admin monitoring was instead done by direct observation of logs and the database, which is beyond the realm of end-user testing.

We switched from a plan to store images in disk storage (with file paths stored in the DB) to one where images are stored in RAM and referenced by Telegram file IDs. This made tests we considered for file storage (e.g., if images were being written to a directory or removed) irrelevant. Instead, we ensured that image processing in memory did not fail for various images and it was safe to use Telegram's file ID later to download images.

We included a data clean routine concept to delete out-of-date records (for privacy, FR-09). While an entire automatic cron job was not coded by project end, we did occasionally hand-clean out-of-date entries. Testing-wise, we ensured that after an avatar was made and time had passed, those images weren't accessible (since we delete them as soon as they're consumed on the remote server). In the next release, a clean-up of the database records routinely would be attempted by checking that ones prior to the threshold are removed.

Testing Details

Equivalence Classes & Paths Tested: We ensured that for each significant module or feature, we considered typical cases, edge cases, and error conditions:

- *Photo Uploads:* Tested valid images vs. invalid inputs (non-images, too few images, duplicate images). Boundary: the 10th image triggers the next step, whereas 9 should not. Also tested extremely large image (to see if Telegram's limits handle it or our bot times out – Telegram compresses images by default, but we tested sending as documents to simulate larger files).

- *Payment (Crypto):* Tested the full happy path: invoice creation -> simulate IPN callback -> check credits. Also tested error path: simulate no callback (user didn't pay) – the user should remain unable to proceed. And partial payment statuses (if IPN comes with "failed" or other statuses, which we logged but essentially treat as no payment).

- *Payment (Stars):* Tested successful payment within Telegram. Also tried the "Cancel" button on the payment invoice to ensure the bot remains functional if user cancels. We verified that multiple payments from the same user only top-up credits (the system uses upsert logic to accumulate credits).

- *Referral:* Created a referral scenario: User A invites User B. User B starts with the link (we manually appended the referral code in the `/start` command during testing). User B pays. Then checked User A's credits increased by 10 and that referral stats updated (1 referral, 1 paid).

- *State Management:* Tested sequences like user tries to upload photos *before* paying (should get a "please pay first" message), or user tries to choose style *before* an avatar is ready (should normally not even see that option, but if forced, it shouldn't proceed). We also tested the "start over" in the middle of a process (e.g., after uploading some photos but before finishing, if the user hits start over, it should reset counts and require payment again).

- *Concurrent Operations:* While we did not have 1000 real users to test with, we simulated multiple parallel sessions by using multiple accounts. We ensured that one user's actions do not interfere with another's data. For instance, two users uploading concurrently maintain separate counts and when their respective 10th photo is uploaded, each triggers their own model generation on the remote server (which can handle sequential or parallel jobs as it has its own scheduling). In practice, if many started at once, they would queue on the single GPU – we monitored that our bot can handle overlapping tasks by queuing them in asyncio and not crashing.

- *Performance:* We measured some timings (not rigorously) – the bot responds to simple commands almost instantly (well under 1 second for /start or menu clicks). The heavy part (training) took ~40 seconds on average, which is within our acceptable range. Generation of each image after training was faster (~5-10 seconds).

- *Security:* We performed basic checks, for example ensuring that a user cannot access another user's data. Since all data is keyed by user ID and the bot logic always uses the current user's context, this was inherently safe. Also, the referral codes are not easily guessable (10-character hash), so one user randomly using another's code is unlikely unless it was shared. We did not formally penetration-test the system, but relying on Telegram for authentication and file storage provides a good baseline security

(the bot never stores actual image files publicly accessible).

Test Results: After implementation, we reran the black-box test cases. As shown in the table in the previous section, almost all test expectations were met:

- The core user journey (payment → upload → generate) worked as intended. We received the correct images in the chosen styles, and the quality was satisfactory given the model used.

- Error handling proved effective: the bot gave meaningful messages for all the error scenarios we tried.

- The referral and credit system updated correctly in tests. We observed the bonus credits being added in real-time after a referred payment.

- One area not fully realized was the admin interface (which was dropped), but this did not affect end-user functionality. Additionally, the automatic data deletion (FR-09) was not fully automated in code, so we did not have a test result for a timed deletion; however, we manually verified that after generating an avatar, if we removed the associated files and records, the system had no dangling references (the bot would simply prompt the user to start over if they tried to use an old session).

- Concurrency tests with a moderate number of parallel users were successful – the bot remained stable and each user's experience was as expected without cross-talk or race conditions in the database. For example, two users could be uploading photos simultaneously and each got correct progress updates and results.

In conclusion, testing confirmed that SnapGenie meets the major functional requirements. The few features that were de-scoped (like the external admin UI) were noted, and the test plan was adjusted accordingly. The system is robust for demonstration purposes, and any potential improvements (like more scalable job queue management or a friendlier admin tool) are left as future enhancements.

Presentation

Instructions:Week 12

Preparation

Project Summary: SnapGenie is a Telegram bot that serves as a personal AI photo studio. It allows a user to transform their selfies into professional, stylized portraits. The user uploads pictures of themselves, the system trains an AI model to “learn” their face, and then the user can generate new images of themselves in various styles (like anime characters, historical figures, etc.). The project showcases an integration of a chatbot interface with heavy backend AI processing and demonstrates how to manage an end-to-end user experience (including payments for monetization and referral-driven growth).

Requirements and Assumptions: In presenting SnapGenie, we will outline the initial requirements and how we addressed them. We assumed a need for both free and premium usage modes; however, for the scope of this project, we implemented a straightforward prepaid credits system (every user goes through a payment to use the service, essentially making all usage “premium” in our demo). We added features beyond the basic brief: specifically, the referral program (not originally mandated but included to encourage user engagement) and multiple payment methods (to lower barriers for users to try the service). We will explain that some initial requirements were adjusted: for example, the admin web dashboard was an assumption that we did not pursue in the final implementation, given time and complexity trade-offs – instead, administrative needs are met via direct data access if needed. Another addition was a conscious focus on privacy (deleting user data after use), which was an implied requirement from “compliance with data protection laws”. We’ll clarify all such assumptions and additions so the audience understands the rationale behind our implementation choices.

Design Choices: We will discuss the key design options we considered and the decisions we made:

- **Architecture choice:** why we chose a Telegram bot with a separate AI service, as opposed to a monolithic app or a mobile app.

- **Technology stack:** our decision to use Python’s asyncio and aiogram vs. other approaches (like a synchronous bot or using a heavier web framework). We’ll weigh the pros and cons: aiogram’s asynchronous nature gave us concurrency and performance benefits, whereas using something like Django could have eased database handling but introduced overhead and wasn’t a natural fit for a bot interface.

- **AI processing approach:** the choice of training a model per user (which gives high quality results) vs. using a generic AI model for all (which would be faster but not person-specific). We opted for the former to meet the quality expectations. We also opted to use an external GPU service rather than local processing or a cloud API like StabilityAI; we’ll mention cost, flexibility, and educational value as factors

(setting up our own mini AI pipeline was instructive and allowed customization).

- **Data management:** the decision to use SQLite and in-memory processing, which simplified development, versus using cloud storage or a larger database. We'll note that SQLite sufficed for our scale and using Telegram's own file storage minimized our need to handle user file storage.

- For each of these, we'll mention alternatives and why we didn't choose them (for instance, not using cloud AI APIs because of cost or dependency, not building a full web app because the interaction is meant to be in Telegram, etc.).

- We will also highlight how adding the **Telegram Stars payment (extension)** affected the design: it required using Telegram's payment platform in addition to an external API. Fortunately, aiogram supports this, so it didn't require a major redesign – we simply added a new handler. Similarly, adding the referral system meant extending our database and logic slightly, but it was designed in a modular way so it slotted in without altering the core flow for non-referred users.

Impact of Extensions: The main extension we implemented was the **referral and rewards system**, as well as the second payment method (Stars). These did not fundamentally change the original design but augmented it:

- The referral system required an extra table and a couple of additional handlers (for sharing invite links and checking stats). It also required hooking into the payment confirmation point to grant bonus credits. Because our design already funneled all payment success logic through a couple of functions, it was easy to insert a call to `giveBonusToInviter` at that point. So the extension was accommodated by the existing design without major refactoring. We'll note that careful upfront design (having a clear payment success path) made this possible.

- The addition of Telegram Stars as a payment method showed the flexibility of the design: our payment interface was abstract enough that we could plug in a second option. It did not disrupt the flow for crypto payments, it just gave an alternate route. In the code, it meant one more handler and making sure the UI presents both options. We'll emphasize that this extension made the system more user-friendly and how we dealt with concurrency in managing two payment systems (essentially by treating them uniformly in terms of credits granted).

- If there were any other extensions (for example, if we imagine future features like filter effects or group photo generation), we can briefly state that the design is modular enough to handle them. But for now, we focus on what we actually added.

Testing Strategy: We will describe how we tested SnapGenie, focusing on black-box testing derived from requirements. This includes how we identified different input categories (images, commands, payments) and their expected outputs. We will mention our equivalence class testing (valid vs invalid inputs) and boundary testing (like exactly 10 photos vs 9). We'll explain how we tested the happy paths (a complete successful flow) as well as various edge cases (network failure, duplicate uploads, etc.). Additionally, we'll mention any tools or methods used in testing – in our case, it was mostly manual testing with multiple Telegram accounts and simulating events. We'll highlight that we observed the system's behavior and compared it against expected results from the test plan, and that the outcomes were positive, with passes on almost all test cases. If any

tests failed initially, we'll note how we debugged and fixed the issues (for example, if we discovered a bug where duplicate photos weren't detected correctly, we patched it and re-tested).

Lessons Learned: Each team member gained significant insights from this comprehensive project:

- *From a software engineering perspective,* we learned the importance of keeping the design flexible. When mid-development changes occurred (like adding a new payment method or dropping a planned feature), having a modular structure meant we could adapt without starting over. We also reinforced the lesson that not every initially planned feature is necessary – sometimes it's prudent to simplify (as we did with the admin panel) to focus on the core value of the project.

- *In terms of programming concepts,* we got hands-on experience with asynchronous programming (using `asyncio` effectively for concurrent operations). This was crucial in making the bot responsive while waiting for long tasks. We also applied object-oriented principles in the design of our data layer (even though Python was used in a more functional way for the implementation, the conceptual design was object-driven).

- We learned about integrating third-party services: from handling webhooks for payments to transferring files over SSH for AI tasks. Each integration taught us to pay attention to details (like security of API keys, error handling for network calls, etc.).

- *Team-wise,* this project highlighted the value of clear division of roles and communication. Our backend developers focused on bot logic and database, the AI developers focused on the model training pipeline and image quality, and the DevOps members ensured the external services (RunPod, etc.) ran smoothly and could be triggered programmatically. We had to coordinate these pieces so that the final system works end-to-end.

- Finally, we appreciated the iterative development process: starting from requirements, writing them down formally, designing the system on paper, then implementing and testing. It helped us catch issues early (for example, thinking through how we'd test something revealed a missing piece in design, which we then added before coding).

Demo Plan (Functionalities to Showcase): In our presentation demo, we plan to **live demonstrate** the core functionalities of SnapGenie:

1. **Onboarding and Payment:** We'll start with a fresh user perspective – showing the `/start` welcome message and going through a payment. (For demo speed, we may use Telegram's test environment or a bypass for the payment, or have pre-granted credits, since waiting for a crypto confirmation might be slow. We might choose the Stars payment since it's instantaneous in test mode.)

2. **Photo Upload:** Next, we will upload a set of photos (we have prepared 10 sample photos of a team member). The audience will see the bot counting the uploads and then indicating that the avatar is being generated.

3. **Avatar Ready & Style Generation:** We'll then show how, after the training wait (we might not actually wait the full training time in the demo – possibly we'll have a model ready or simulate completion for the sake of time), the bot allows style selection. We will choose a style (say, "Anime") and within moments,

the bot will respond with a generated image. We will repeat with another style (or a custom prompt) to show versatility. The returned images will be visible in the Telegram chat for everyone to see the effect.

4. **Referral Feature:** We plan to demonstrate the referral briefly – for example, one of us will use another's referral link to start the bot. We won't go through another full payment, but we can show that the referral code gets acknowledged (maybe by printing a debug message or showing the referral count increment on the inviter's side).

5. **Other Commands:** If time permits, we'll show the "Balance" command (the bot will reply with remaining credits) and possibly the "Start Over" function (though we may not actually upload another set in the demo, just show that it resets the state).

6. We will also talk through what's happening behind the scenes for each step, tying it back to our architecture (for example, "now the bot is calling the remote server to train the model…" etc.).

By clearly dividing responsibilities and practicing our segments, our team will deliver a well-rounded presentation that not only shows a working software product but also communicates the engineering thought process behind it. Each member speaking to their part underscores the collaboration that went into SnapGenie. We are excited to demonstrate SnapGenie and share what we learned in building it.