НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

«На правах рукопису»
УДК ___004.43__

До захисту допущено:
В. О. Завідувача кафедри
_____Михайло НОВОТАРСЬКИЙ
«___»_____2025 р.

Магістерська дисертація

на здобуття ступеня магістра
за освітньо-професійною програмою «Інженерія програмного забезпечення комп'ютерних систем» зі спеціальності 121 «Інженерія програмного забезпечення» на тему: «Спеціалізована JavaScript бібліотека для створення персональних асистентів у месенджері Signal»

Виконала:
студентка ІІ курсу, групи ІМ-41мп
Мошенська Олеся Олегівна _____

Керівник:
доц. каф. ОТ, к.т.н., с.н.с.
Долголенко Олександр Миколайович _____

Консультант з нормоконтролю:
проф. каф. ОТ, д.т.н., професор
Жабін Валерій Іванович

Рецензент:
доц., к.т.н., доц. кафедри СПСКС ФПМ
Щербина Олександр Андрійович

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2025 рік

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет *Інформатики та обчислювальної техніки*
(повна назва)

Кафедра *Обчислювальної техніки*
(повна назва)

Рівень вищої освіти – другий (магістерський)

Спеціальність *121. Інженерія програмного забезпечення*
(код і назва)

Освітньо-професійна програма *121. Інженерія програмного забезпечення комп'ютерних систем*
(код і назва)

ЗАТВЕРДЖУЮ
В. о. завідувача кафедри
_____Михайло НОВОТАРСЬКИЙ
(підпис)
«____» 2025 р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Мошенській Олесі Олегівні
(прізвище, ім'я, по батькові)

1. Тема дисертації Спеціалізована JavaScript бібліотека для створення персональних асистентів у месенджері Signal

Науковий керівник дисертації Долголенко Олександр Миколайович, доцент, кандидат технічних наук, старший науковий співробітник
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «06» листопада 2025 р. № 4841-с

2. Строк подання студентом дисертації 30.11.2025

3. Об'єкт дослідження процес розроблення програмного забезпечення персональних асистентів в умовах відсутності відкритого API месенджера Signal.

4. Предмет дослідження методи управління стратегією діалогу та формування відповіді персональних асистентів.

5. Перелік завдань, які потрібно розробити: проаналізувати предметну область, існуючі рішення та сформулювати вимоги до бібліотеки, дослідити та обрати архітектуру і технології для реалізації бібліотеки, розробити програмні компоненти бібліотеки, провести тестування розробленого програмного засобу.

6. Консультанти розділів дисертації:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1	Долголенко О. М., проф. каф. ОТ		
2	Долголенко О. М., проф. каф. ОТ		
3	Долголенко О. М., проф. каф. ОТ		
4	Долголенко О. М., проф. каф. ОТ		

7. Дата видачі завдання 1.09.2025

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Строк виконання етапів дисертації	Примітка
1	Затвердження теми дослідження. Визначення предмету дослідження	1.09.25-7.09.25	
2	Дослідження існуючих рішень та проблем конструювання трафіка в досліджуваній області	8.09.25-21.09.25	
3	Побудова та обґрунтування архітектурного рішення	22.09.25-28.09.25	
4	Моделювання генерації трафіка в SDN мережі	29.09.25-05.10.25	
5	Реалізація системи безперервної потокової передачі повідомлень	06.10.25-19.10.25	
5	Розробка процесу обробки, агрегації та акумуляції мережевих даних	20.10.25-09.11.25	
6	Тестування моделі та аналіз її ефективності	10.11.25-13.11.25	
7	Розробка стартап-проекту	14.11.25-18.11.25	
8	Оформлення магістерської дисертації	19.11.25-24.11.25	

Студент Олеся МОШЕНСЬКА _____
(підпис)

Науковий керівник дисертації Олександр ДОЛГОЛЕНКО _____
(підпис)

РЕФЕРАТ
на магістерську дисертацію

виконану на тему: Спеціалізована JavaScript бібліотека для створення персональних асистентів в месенджері Signal

студенткою: Мошенською Олесею Олегівною

Актуальність теми. Месенджери стали зручною платформою для автоматизації бізнесу та рутинних завдань, де персональні асистенти виступають основними засобами взаємодії. Проте Signal, на відміну від інших платформ, не має офіційного API, що створює технічні обмеження. Тому користувачі, які обирають цей месенджер заради безпеки, позбавлені можливості створювання повноцінних асистентів. Розробка спеціалізованого рішення усуває цю проблему та відкриває шлях до побудови корпоративних і приватних асистентів.

Мета і завдання дослідження. Метою роботи є розробка спеціалізованих програмних засобів, що спростять програмну взаємодію з платформою Signal та нададуть зручний інструментарій для створення персональних асистентів у цьому месенджері. Для досягнення цієї мети були поставлені та вирішені наступні завдання:

1. Проаналізовано предметну область, технічні обмеження, що накладає платформа Signal, та оцінені існуючі програмні рішення.
2. Упорядковані функціональні та нефункціональні вимоги до бібліотеки.
3. Спроєктовані та реалізовані базові компоненти бібліотеки.
4. Протестовано бібліотеку та практично перевірено роботу персонального асистента, створеного з використанням її засобів.

Об’єкт дослідження. Об’єктом дослідження є процес розроблення програмного забезпечення персональних асистентів в умовах відсутності відкритого API месенджера Signal.

Предмет дослідження. Предметом дослідження є методи управління стратегією діалогу та формування відповіді персональних асистентів.

Методи досліджень. Використано порівняльний аналіз для оцінювання існуючих рішень за критеріями функціональної повноти, теорію графів та автоматів.

Наукова новизна. Вперше для даної предметної області запропоновано підхід до керування діалогом на основі скінченних автоматів із збереженням контексту розмови, часовими обмеженнями та динамічною зміною стану, що дозволяє перейти від атомарних команд до реалізації складної логіки розгалуження діалогу.

Практичне значення одержаних результатів. Бібліотека є мінімально життєздатним продуктом (MVP), що побудована на відкритій до розширень архітектурі та включає основоположні інструменти реалізації персональних асистентів. Робота є міцним фундаментом для подальшого розвитку бібліотеки у повноцінний, конкурентоспроможний інструмент.

Особистий внесок здобувача. Теоретичні та практичні висновки, викладені у дисертації, отримані автором самостійно і є його особистим внеском. Визначення напрямку, мети та завдань дослідження проводилося спільно з науковим керівником.

Структура та обсяг дисертації. Робота складається зі вступу та чотирьох розділів. Загальний обсяг роботи: 134 аркушів, 63 рисунків, 27 таблиць. При підготовці використовувалася література з 29 джерел.

Ключові слова. Платформа обміну повідомленнями, Signal, персональний асистент, діалоговий помічник, бібліотека, JavaScript, управління виконанням діалогу, діалогові сценарії.

ABSTRACT

for master's thesis

on the topic: Specialized JavaScript Library for Creating Personal Assistants in Signal Messenger

by student: Moshenska Olesia Olehivna

Relevance of the topic. Messengers have become a convenient platform for business automation and routine tasks, where personal assistants serve as primary means of interaction. However, Signal, unlike other platforms, does not have an official API, which creates technical limitations. Therefore, users who choose this messenger for security reasons are deprived of the ability to create full-fledged assistants. The development of a specialized solution eliminates this problem and opens the way to building corporate and private assistants.

The purpose and objectives of the research. The aim of this work is to develop specialized software tools that will simplify programmatic interaction with the Signal platform and provide convenient toolkit for creating personal assistants in this messenger. To achieve this goal, the following tasks were set and accomplished:

1. The subject area, technical limitations imposed by the Signal platform were analyzed, and existing software solutions were evaluated.
2. Functional and non-functional requirements for the library were systematized.
3. Basic components of the library were designed and implemented.
4. The library was tested and the operation of a personal assistant created using its tools was practically verified.

Object of research. The object of research is the process of developing personal assistant software in the absence of an open API for the Signal messenger.

Subject of research. The subject of research is methods of dialogue strategy management and response formation for personal assistants.

Research methods. Comparative analysis was used to evaluate existing solutions based on functional completeness criteria, graph theory and automata theory.

Scientific novelty. For the first time in this subject area, an approach to dialogue management based on finite state machines with conversation context preservation,

time constraints and dynamic state changes has been proposed, which allows moving from atomic commands to implementing complex dialogue branching logic.

Practical significance of the results obtained. The library is a minimum viable product (MVP) built on an architecture open to extensions and includes fundamental tools for implementing personal assistants. The work provides a solid foundation for further development of the library into a fully-fledged, competitive tool.

Master's personal contribution. The theoretical and practical conclusions presented in the dissertation were obtained by the author independently and represent their personal contribution. The determination of the research direction, objectives and tasks was carried out jointly with the scientific supervisor.

Structure and scope of the dissertation. The work consists of an introduction and four chapters. Total volume of the work: 134 pages, 63 figures, 27 tables. Literature from 29 sources was used in the preparation.

Keywords. Messaging platform, Signal, personal assistant, dialogue system, library, JavaScript, dialogue context management, dialogue scenarios.

ЗМІСТ

ВСТУП

В умовах стрімкої цифровізації суспільства платформи обміну миттєвими повідомленнями трансформувалися з простих засобів комунікації у багатофункціональні сервіси. Це створює попит на автоматизацію взаємодії через інтелектуальних агентів, зокрема чат-ботів та звичайних персональних асистентів, які знаходять дедалі ширше застосування як у бізнес-процесах, так і у оптимізації повсякденних завдань. На цьому тлі месенджер Signal посідає унікальне місце, пропонуючи особливу політику безпеки. Такий акцент на конфіденційність та захист даних приваблює значний сегмент користувачів та організацій, які цінують приватність своїх даних.

Проте незважаючи на зростаючий інтерес до платформи, її потенціал для автоматизації залишається значною мірою нереалізованим. Головною перешкодою є відсутність офіційного інтерфейсу програмування застосунків. Як наслідок, високий попит на безпечних персональних асистентів стикається з відсутністю повноцінного, надійного та зручного інструментарію для їх розробки.

Більшість розробок зосереджені навколо платформ з відкритими API, таких як Telegram, Discord чи Slack, де існує розвинена інфраструктура. Ті ж інструменти, що існують для Signal, часто є складними у налаштуванні або пропонують лише мінімально необхідну функціональність. Це створює потребу у розробці спеціалізованого програмного рішення, орієнтованого саме на Signal.

Магістерська робота спрямована на вирішення зазначеної проблеми шляхом проєктування та реалізації спеціалізованої бібліотеки мовою JavaScript. Основна ідея полягає у спрощенні процесу розроблення персональних асистентів для Signal, надаючи зручний програмний інтерфейс для побудови складної логіки взаємодії з користувачами месенджера, при цьому приховуючи всі технічні особливості роботи з платформою. Це відкриє нові можливості для бізнесу та приватних користувачів щодо автоматизації завдань у середовищі, що виключає доступ третіх сторін до даних.

РОЗДІЛ 1
ДОСЛІДЖЕННЯ ТЕХНОЛОГІЙ ТА ОСОБЛИВОСТЕЙ РОЗРОБЛЕННЯ ПЕРСОНАЛЬНИХ АСИСТЕНТІВ У МЕСЕНДЖЕРІ SIGNAL

1.1. Тенденції розвитку захищених комунікацій та засобів автоматизації

Сучасний стан розвитку інформаційних технологій характеризується двома, на перший погляд, суперечливими тенденціями: масовим попитом на автоматизацію комунікації та виконання рутинних завдань через діалогових помічників, і зростаючими вимогами користувачів до конфіденційності даних.

Ринок діалогових систем демонструє стабільне зростання. За даними аналітичних звітів [1], обсяг ринку чат-ботів у 2025 році оцінюється у 9.30 мільярда доларів США, і прогнозується, що до 2030 року він досягне 27.07 мільярда із середньорічним темпом зростання близько 23.8%. При цьому одним із найвпливовіших чинників, що поряд із прогресом у сфері LLM стимулюють розвиток галузі, є стрімке розширення аудиторії месенджерів (рис. 1.1).

DRIVER	(~) % IMPACT ON CAGR FORECAST	GEOGRAPHIC RELEVANCE	IMPACT TIMELINE
Explosion of messaging-app user base	+4.2%	Global, with APAC leading adoption	Medium term (2-4 years)
Breakthroughs in large-language-model (LLM) NLP	+5.8%	North America and the EU core, expanding globally	Short term (≤ 2 years)
24/7 customer-support cost pressure	+3.9%	Global, particularly in high-labor-cost regions	Short term (≤ 2 years)
Self-service mandates in digital CX strategies	+3.1%	North America and the EU, spreading to APAC	Medium term (2-4 years)
Voice-first and multimodal bot convergence	+2.7%	Global, with early adoption in North America	Long term (≥ 4 years)
LLM-powered internal knowledge automation	+3.3%	Enterprise-focused, primarily North America and the EU	Medium term (2-4 years)

Рисунок 1.1. Глобальні чинники, що формують попит на чат-боти [1]

Бізнес та користувачі дедалі частіше розглядають месенджери не просто як засіб спілкування, а як платформу для отримання послуг чи автоматизації

повсякденних завдань. Паралельно з цим відбувається міграція аудиторії до більш захищених платформ, що посилилася після інциденту 2022 року з масовим витоком даних активних користувачів через API-скрапінг месенджера WhatsApp, що належить Meta [2].

Серед найпопулярніших та найбільш захищених месенджерів виділяється Signal (рис. 1.2), який позиціонується як еталон приватності, хоча й залишається доволі нішевим порівняно з іншими платформами, такими як Telegram.

Messenger Filter Table	Google Messages	Element (Matrix)	Signal	Telegram	Threema	Wire
App collects customers' data?	Yes (Difficult to assess given the app is integrated into Google's greater ecosystem)	Contact info / contacts / identifiers / diagnostics / user content (Contact info not sent when using anonymously)	Contact Info	Contact info / contacts / identifiers	Contact info / identifiers / diagnostics (Contact info not sent when using anonymously)	Contact info / identifiers / usage data / diagnostics
User data and/or metadata sent to parent company and/or third parties?	Yes	No (User data is sent to a third party if a payment is made)	Minimal (Mandatory mobile number sent to third party for registration & recovery)	Yes	No (Optional mobile number sent to third party for registration)	Yes
Is encryption turned on by default?	Yes	Yes	Yes	No	Yes	Yes
Cryptographic primitives	Curve25519 / AES-256 / HMAC-SHA256	Curve25519 / AES-256 / HMAC-SHA256	Curve25519 & Kyber-1024 / AES-256 / HMAC-SHA256/512	RSA 2048 / AES 256 / SHA-256	Curve25519 256 / XSalsa20 256 / Poly1305-AES 128	Curve25519 / ChaCha20 / HMAC-SHA256
Can you sign up to the app anonymously?	No	Yes	No	No	Yes	No
Can messages be read by the company?	No	No	No	Yes	No	No
Does the app enforce perfect forward secrecy?	Yes	Yes	Yes	No (session keys do change after being used 100 times)	Yes	Yes
Does the app encrypt metadata?	No		Yes	No	Yes	Mostly

Рисунок 1.2. Порівняння рівнів приватності та захищеності в месенджерах [3]

Signal – це кросплатформовий месенджер для обміну миттєвими повідомленнями, розроблений організацією Signal Foundation. Платформа бере свій початок із об'єднання проєктів RedPhone та TextSecure, а її програмне забезпечення стало основою для функцій наскрізного шифрування у таких сервісах, як WhatsApp, Facebook Messenger, Skype та Google Allo. Станом на 2024 рік кількість активних користувачів платформи оцінюється у 70 мільйонів, а загальна кількість завантажень перевищила 220 мільйонів (рис. 1.3) [4].

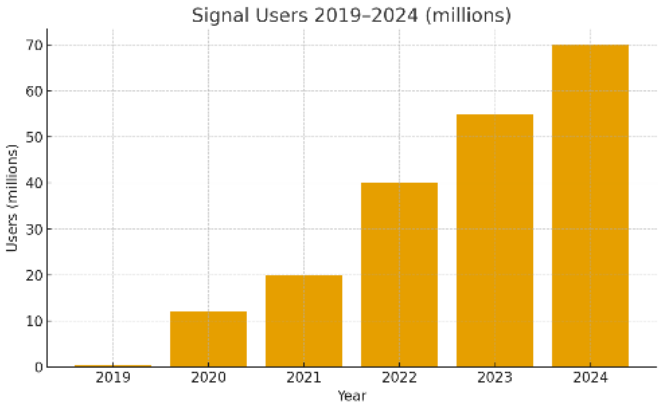


Рисунок 1.3. Графік приросту користувачів Signal протягом 2019-2024 років

Територіально трафік Signal розподілений досить нерівномірно – найбільше його використовують у Німеччині (19.9%) і США (17.35%), що пов’язано зі строгими регламентами захисту даних у цих країнах, а також в Україні (5.63%) [5], де месенджер широко застосовується як засіб захищеного зв’язку в умовах повномасштабної війни та постійного ризику перехоплення інформації (рис. 1.4).

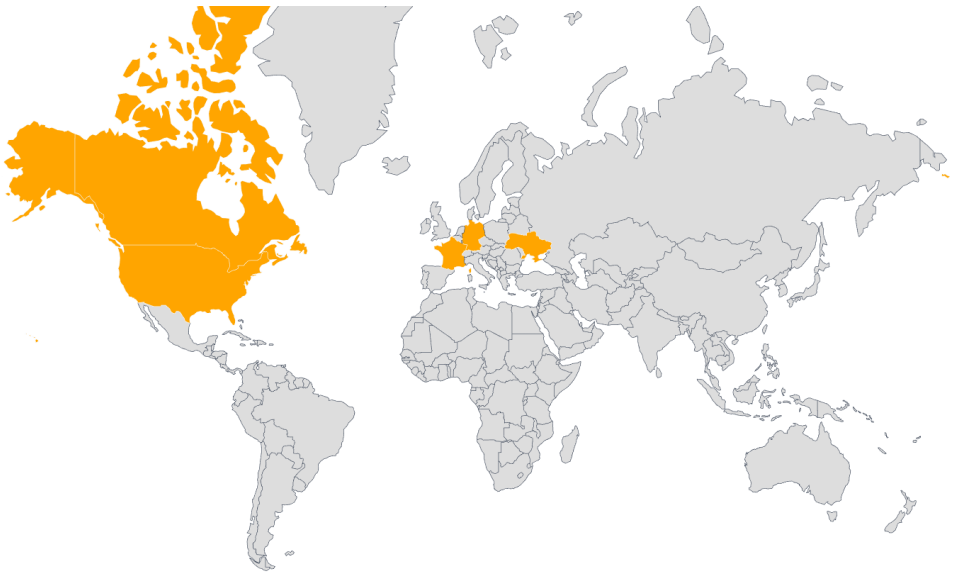


Рисунок 1.4. Географічний розподіл користувачів Signal [5]

Далі пропонується розглянути, чому саме цей месенджер посідає позицію одного з найзахищеніших рішень, і які можливості для користувачів відкриває створення персональних асистентів у його середовищі.

1.2. Особливості месенджера Signal як платформи для персональних асистентів

Функціонально Signal надає стандартний набір можливостей: обмін текстовими та голосовими повідомленнями в приватних і групових чатах, здійснення аудіо- та відеодзвінків, передача файлів. Однак головна відмінність Signal полягає саме безпеці, а не в розширенні функціональності.

1.2.1. Безпекова модель Signal

Клієнтські застосунки для основних платформ, серверна частина месенджера та інше програмне забезпечення розповсюджуються з відкритим вихідним кодом [6]. В основі месенджера лежить власний криптографічний протокол, який забезпечує E2E шифрування та використовує комбінацію X3DH (Curve25519), Double Ratchet Algorithm, AES-256 та HMAC-SHA256, що вважається перевіреним і надійним рішенням, рекомендованим багатьма незалежними дослідниками [2, 7, 8].

Хоча E2E-шифрування сьогодні вважається стандартом для месенджерів, важливо врахувати, що попри відповідні заяви, у більшості сервісів воно не активоване за замовчуванням. Наприклад, для Facebook Messenger та Telegram воно насправді працює лише в «секретних чатах». Крім того, за MTProto у Telegram зміна ключів ініціюється лише коли вони були у використанні для більше ніж 100 повідомлень або понад один тиждень. На противагу, Signal працює на принципах прямої та майбутньої секретності, оновлюючи ключі для кожного окремого повідомлення. Тобто, компрометація поточного ключа не дозволяє зловмиснику розшифрувати жодного повідомлення з минулого і, в більшості випадків, жодного повідомлення з майбутнього (до отримання наступного повідомлення від співрозмовника).

Іншою особливістю Signal є те, що сервери виконують виключно функцію ретранслятора, передаючи зашифровані повідомлення без можливості доступу до їхнього вмісту. Вся історія листування, ключі шифрування та контактні дані зберігаються виключно локально на пристроях користувачів [2, 8].

Такий рівень захисту відкриває можливість використання Signal у сценаріях, де конфіденційність є вимогою та визначає допустимість самої системи. Це охоплює як корпоративні сфери – фінансові консультації, юридичну підтримку, внутрішні комунікації з даними, доступ до яких має бути обмеженим, – так і приватні побутові застосунки, включно зі сповіщеннями від систем розумного дому чи звичайних персональних помічників.

1.2.2. Технічні виклики роботи з месенджером

Одночасно з цим, такий жорсткий захист створює проблему для реалізації відкритого API. У месенджерах, де сервер зберігає історію діалогу та ключі на овермих серверах, боти мають можливість отримувати та надсилати повідомлення через HTTP-запити. У випадку Signal сервер не володіє ключами для розшифрування й не має жодної інформації про попередні повідомлення чи історію листування. До того ж, як неприбуткова організація, Signal Foundation не має передбачає витрату ресурсів на розробку та підтримку повноцінного відкритого API [9].

Відсутність централізованого доступу до даних та необхідність виконання складних криптографічних операцій на стороні клієнта призводять до того, що будь-який програмний засіб для автоматизації Signal повинен фактично виконувати роботу повноцінного клієнтського застосунку. Крім цього, існує низка й інших специфічних викликів:

1. Складність автентифікації. На відміну від платформ із API-токенами для ботів, система повинна пройти повний процес реєстрації та верифікації, включаючи генерацію ключів і підтвердження номеру телефону.
2. Децентралізована обробка подій. Оскільки сервери Signal виконують лише транспортні функції, отримання та обробка всіх вхідних повідомлень, а також створення сесійних даних, відбувається повністю на клієнті й потребує постійної роботи в режимі реального часу.
3. Обмеження на відправку великих обсягів повідомлень. Сервери Signal після масової міграції користувачів із WhatsApp застосовують тротлінг та

додатковий захист від спаму [4], тому на клієнті необхідно керувати чергою, частотою та повторними спробами відправки самостійно.

4. Необхідність оновлень клієнта. Signal підтримує політику примусового оновлення клієнтів кожні три місяці з можливістю внесення кардинальних змін у протокол, що створює ризики несподіваної втрати функціональності та вимагає постійного моніторингу оновлень.

Отже, месенджер Signal, який забезпечує високий рівень приватності для користувачів, одночасно утворює додаткові технічні складності, що обґрунтовує необхідність створення спеціалізованого інструменту, який би приховував складність взаємодії з платформою та надавав зручний інтерфейс для побудови персональних асистентів.

1.2.3. Signal-cli як інструмент для автоматизації

Відсутність офіційного API для взаємодії з Signal обумовила розвиток альтернативних технічних рішень, серед яких найпопулярнішим є проєкт Signal-cli. Signal-cli – це неофіційний, але широко використовуваний і активно підтримуваний спільнотою інструмент командного рядка, який реалізує клієнт Signal для основних операційних систем (Linux, macOS, Windows) [10].

Signal-cli функціонує як самостійний застосунок з двома основними режимами роботи. У командному режимі він дозволяє виконувати дискретні операції через окремі виклики командного рядка (реєстрацію облікових записів, надсилання повідомлень, управління групами). Альтернативно, у режимі демона він може працювати як фонова служба, отримуючи вхідні повідомлення та надаючи їх у JSON-форматі через стандартний вивід або D-Bus інтерфейс.

Signal-cli також вирішує головну проблему програмного доступу до месенджера Signal, беручи на себе всю складність криптографічної роботи – від взаємодії з протоколом до управління ключами шифрування та автентифікації на серверах [9]. До того ж, активна підтримка як самого розробника, так і спільноти вирішує проблему обмеженого терміну сумісності клієнта, гарантуючи своєчасні оновлення відповідно до змін у протоколі.

Водночас Signal-cli в першу чергу призначений для використання з метою повідомлення адміністраторів про важливі події [10], а не є цілісним інструментом для розробки ботів. Взаємодія з ним зводиться до надсилання інструкцій та обробки відповідей. Проте цей інструмент вартий уваги, оскільки може слугувати надійним фундаментом для побудови власного рішення, як це демонструють програмні засоби, що розглядатимуться далі.

1.3. Теоретичні основи та класифікація персональних асистентів

Перш ніж оцінювати технічний стан галузі необхідно визначити, що в межах цієї роботи мається на увазі під персональним асистентом. Це допоможе узгодити терміни та окреслити коло діалогових систем, на які буде орієнтована бібліотека.

У сучасній науковій літературі, зокрема у роботах [11, 12], термін «персональний асистент» (або чат-бот, діалогова система) трактується досить широко. Дослідження фокусуються на огляді чат-ботів як агентів на основі ШІ, але у загальному вигляді – це програмне забезпечення, здатне взаємодіяти з людиною природною мовою через текстовий або голосовий інтерфейс для виконання певних завдань або надання інформації. Ідея таких систем сягає 1950-х років і тесту Тюрінга, але сучасний етап розвитку характеризується переходом від простих шаблонних рішень до інтелектуальних агентів, здатних підтримувати складний контекст розмови.

Дослідники класифікують персональних асистентів за різними критеріями, зокрема за стратегією управління діалогом [11]:

1. Системи, засновані на скінченних станах (або графах) – це найпростіший тип, де діалог складається з послідовності заздалегідь визначених кроків або станів. Цей підхід зазвичай використовується у більшості комерційно доступних розмовних діалогових систем.
2. Системи на основі фреймів – це більш гнучкий підхід до управління, де метою діалогу є заповнення певної структури даних (шаблону чи

форми). Потік діалогу не є заздалегідь визначеним, а залежить від змісту введених користувачем даних та інформації, яку має отримати система.

3. Агентні системи, що використовують методи штучного інтелекту для моделювання поведінки, намірів та переконань. Такі асистенти здатні до змішаної ініціативи, коли напрямок розмови може змінювати як користувач, так і система, вирішуючи складні завдання спільно.

У дослідженні [12] надається більш широка класифікація, зокрема за доменом знань, типом відносин, основною метою, дозволами, каналами зв'язку (текст, голос, зображення тощо). Утім, зазначені ознаки стосуються переважно прикладного використання чат-ботів, тоді як у межах цієї роботи релевантним є метод формування відповіді, який може бути:

1. На основі правил, що є основою для систем на скінченних автоматах та частково систем на основі фреймів. Відповідь обирається із заздалегідь визначеного набору, зіставляючи ввід користувача із правилом-шаблоном.
2. На основі вибірки, де застосовується нейронна мережа для оцінювання релевантності кожної потенційної відповіді, якій присвоюється певний бал відповідності поточному контексту й обирається варіант із найвищою оцінкою.
3. На основі динамічного синтезу відповіді, зазвичай за допомогою мовної моделі, яка генерує текст опираючись на попередній контекст, а не обмежується наперед заданим набором фраз.

Для цілей даної роботи найбільш доцільною видається реалізація функціоналу для побудови графових систем з формуванням відповіді на основі правил, оскільки це покриє більшість випадків використання діалогових помічників. Водночас, варто передбачити простір для майбутнього покращення через обрання відповідей на основі вибірки.

1.4. Огляд існуючих програмних засобів

Під час попереднього огляду рішень було знайдено значну кількість проєктів різного рівня завершеності та доступої функціональності, що свідчить про існування суттєвої потреби у інструментах автоматизації для даної платформи. Для детального розгляду було відібрано проєкти, що найбільш відповідають критеріям активної підтримки, функціональної повноти та релевантності до поставлених завдань роботи.

1.4.1. Фреймворк Signalbot

Signalbot – це мікрофреймворк на мові Python, орієнтований на створення ботів у месенджері Signal із використанням команд на основі декораторів [13]. Команди визначаються у вигляді класів через @triggered для активації за точною відповідністю ключових слів або @regex_triggered для активації за шаблонами регулярних виразів. Кожна команда отримує об'єкт контексту повідомлення, який також надає API для швидкого формування відповідей: send для відправлення нових повідомлень, reply для відповіді з цитуванням адресата, react для реакцій емодзі, а також імітацію набору тексту (рис. 1.5).

```python
class PingCommand(Command):
    @triggered("Ping")
    async def handle(self, c: Context) -> None:
        await c.send("Pong")
```

Рисунок 1.5. Приклад оголошення команди засобами фреймворка Signalbot [13]

Загалом, у проєкті реалізовано базовий набір можливостей для автоматизації взаємодії з користувачами Signal. Окрім вже згаданих функцій, фреймворк дозволяє редагувати і видаляти повідомлення, керувати налаштуваннями груп, опрацьовувати вкладення, планувати виконання завдань за розкладом. Крім цього, у ньому надаються інтерфейси для збереження

тимчасових сесійних даних: у пам’яті застосунку, що використовується за замовчуванням, у реляційній СУБД SQLite та у розподіленому сховищі Redis [14].

Функціонування фреймворку залежить від наявності запущеного сервера Signal-cli-rest-api (розглядається далі), який, у свою чергу, функціонує як обгортка над консольною утилітою Signal-cli, яка також повинна працювати у режимі фонового процесу та бути доступною через порт або сокет. З точки зору зручності використання для кінцевого користувача такий підхід є досить обтяжливим, адже потрібно встановити та налаштувати три окремих програмних засоби, забезпечити їхню версійну сумісність та WebSocket-комунікацію між процесами.

Крім того, аби пройти процедуру реєстрації через сканування QR-коду, Signal-cli потрібно спочатку запустити як консольний застосунок, а вже потім як демон для роботи.

1.4.2. Модуль Signal-bot

Signal-bot – це ще один програмний засіб, реалізований мовою Python, відмінністю якого є пряма взаємодія з Signal-cli без використання проміжного REST API шару. Рішення пропонує роботу з Signal-cli через механізм дочірніх процесів у межах одного середовища виконання або через мережевий протокол TCP [15].

Проєкт організовано за подієвою моделлю: для кожного типу вхідних даних можна визначити окремі функції-обробники (рис. 1.6).

```
async def crabby_callback(signal: Signal, context: Context, message: DataMessage) -> bool:
        to = context[1]
        await signal.send_reaction(message, "🦀")
        await signal.send_message(to, f"Sorry, just feeling a little crabby, {message.sender_name}.",
                                  args=SendMessageArgs(mention=[f"37:{len(message.sender_name)}:{messag
        return True
```

Рисунок 1.6. Функція обробки повідомлення з реакцією та відповіддю у Signal-bot [15]

Також, надаються методи для реакції на різні види повідомлень – від точного збігу команди до наявності ключових слів чи згадки ідентифікатора бота. У проєкті також представлено сутність «особистостей», за допомогою якої можна налаштувати окремі правила обробки вхідних данних від певних учасників діалогу чи груп.

Набір реалізованих функцій покриває основні потреби, проте не охоплює повний спектр можливостей, доступних через Signal-cli, таких як керування груповими налаштуваннями або обробка вкладень. Окрім цього його функціональність нічим не відрізняється від фреймворку Signalbot, хоча API виглядає дещо складнішим.

1.4.3. Бібілотека Signal-cli-rest-api

Signal-cli-rest-api – це REST API обгортка над Signal-cli, реалізована мовою Go [16]. На відміну від попередньо розглянутих рішень, це не бібліотека чи фреймворк для розробки ботів, а проміжний шар, який надає HTTP-інтерфейс до функціональності Signal-cli. Низка інших проектів, зокрема Signalbot та Signal-rest-ts, використовують його як обов'язкову залежність для взаємодії з Signal.

Проєкт підтримує взаємодію з Signal-cli у трьох режимах виконання:

- У режимі normal Signal-cli викликається для кожного REST API запиту, що кожного разу запускає віртуальну машину Java та, відповідно, має найповільнішу швидкість роботи.
- Native використовує попередньо скомпільований бінарний файл засобами GraalVM, завдяки чому затримка є суттєво нижчою та пам’ять споживається менше.
- У режимі json-rpc запускається єдиний екземпляр Signal-cli як фоновий процес, що є найшвидшим варіантом, проте потребує більшого обсягу оперативної пам'яті.

Функціональне покриття включає повний спектр операцій з месенджером Signal, починаючи від реєстрації облікових записів та обробку капчі до деталей форматування тексту за синтаксисом, схожим до Markdown.

Проєкт надає образи Docker, повністю задокументований через OpenAPI (Swagger) та підтримує розширення через плагіни для реєстрації додаткових ендпоінтів без необхідності створення форка.

Хоча й рішення не має спеціалізованих інструментів для створення персональних асистентів, воно забезпечує стандартизований інтерфейс, який можна використовувати у різних середовищах, і для випадків, коли необхідно швидко налагодити програмну взаємодію з месенджером без дослідження деталей роботи Signal-cli чи бібліотек самого Signal, може бути прийнятним компромісом.

1.4.4. Бібліотека Signal-rest-ts

Signal-rest-ts – це TypeScript обгортка над Signal-cli-rest-api, яка, на відміну від останнього, надає програмний інтерфейс для розробки автоматизованих агентів у месенджері Signal. Рішення реалізоване за принципами сервісно-орієнтованої архітектури: центральний клас SignalClient надає доступ до спеціалізованих сервісів для роботи з обліковими записами, повідомленнями та групами [17].

Для обробки вхідних повідомлень бібліотека також надає об'єкт контексту, який передається у зареєстровані обробники подій. Вибір конкретного обробника здійснюється на основі співставлення тексту повідомлення з регулярними виразами (рис. 1.7).

```
const signal = new SignalClient("http://localhost:8080");
const accounts = await signal.account().getAccounts();

signal.receive().registerHandler(accounts[0], /^(ha){2,}/, async (context) => {
  console.log(context.sourceUuid + " -> " + context.message);
  context.reply("What's so funny?");
});

signal.receive().startReceiving(accounts[0]);
```

Рисунок 1.7. Визначення обробників повідомлень у Signal-rest-ts [17]

Важливою перевагою є підтримка кількох середовищ виконання: як серверне середовище Node.js, так і браузер через окремий пакет. Бібліотека

також надає функціонал для роботи з вкладеннями, дозволяючи ботам як приймати, так і відправляти файли різних типів.

Попри наявність мінімального інструментарію для обробки повідомлень користувачів Signal, рішення не позиціонується як повноцінна платформа для створення персональних асистентів.

1.4.5. Порівняння розглянутих рішень

Тож, існуючі інструменти для Signal характеризуються значним різноманіттям за структурою та функціональністю. Для об'єктивного порівняння цих інструментів доцільно застосувати єдину систему критеріїв. Такою основою слугує визначення персонального асистента з п. 1.3, а також деякі технічні виклики, притаманні месенджеру Signal як середовищу виконання, розглянуті у п.п. 1.2.2. Узагальнені результати аналізу представлені у таблиці 1.1.

Таблиця 1.1

Порівняльна таблиця існуючих рішень

Рішення / Критерій	Signalbot	Signal-bot	Signal-cli-rest-api	Signal-rest-ts
Мова реалізації	Python	Python	Go	TypeScript
Залежності для роботи з Signal	Signal-cli-rest-api, Signal-cli	Signal-cli	Signal-cli	Signal-cli-rest-api, Signal-cli
Управління криптографією	Делегується до Signal-cli	Делегується до Signal-cli	Делегується до Signal-cli	Делегується до Signal-cli
Спеціалізовані засоби для роботи з вхідними даними	Контекст повідомлень та швидкі методи відповіді	Контекст повідомлень та швидкі методи відповіді	Відсутні	Контекст повідомлень та швидкі методи відповіді

Таблиця 1.1 (продовження)

Порівняльна таблиця існуючих рішень

Рішення / Критерій	Signalbot	Signal-bot	Signal-cli-rest-api	Signal-rest-ts
Модель взаємодії з користувачем	Команди на основі шаблонів, ключові слова	Команди на основі шаблонів, подієві обробники, згадки	Відсутня	Команди на основі шаблонів
Стратегія управління діалогом	Відсутня	Відсутня	Відсутня	Відсутня
Підтримка зовнішніх сховищ	SQLite, Redis	Відсутня	Відсутня	Відсутня
Рівень покриття тестами	Юніт-тестування основних функцій	Тестування відсутнє	Юніт-тести окремої функціональності	Юніт- та інтеграційні тести всіх сервісів
Інструментарій тестування клієнтського коду	Емуляція обміну повідомленнями	Відсутній	Відстуній	Відсутній
Документація та приклади	README, приклади	README, приклади, ReadTheDocs	README, Swagger	README, приклади, TypeDoc

Тож, кожне рішення має різний рівень функціональної повноти та якості реалізації, проте жодне із них не надає достатньої кількості зручних засобів для створення повноцінного персонального асистента.

Найсуттєвішою прогалиною є відсутність вбудованих спеціалізованих інструментів управління діалогом в усіх рішеннях. Відповідно, структура взаємодії обмежується атомарними командами чи простими подієвими обробниками, оскільки реалізація більш складної взаємодії потребує послідовного збору та збереження інформації від користувача. Тож, розглянуті інструменти змушують або обмежуватися простими ізольованими взаємодіями, або самостійно імплементувати всю логіку управління потоком діалогу.

Отже, простежується необхідність у розробці спеціалізованого іструменту, який би системно вирішував виявлені проблеми та надавав цілісне рішення для створення персональних асистентів у месенджері Signal.

ВИСНОВОК ДО ПЕРШОГО РОЗДІЛУ

Проєктування спеціалізованої бібліотеки для створення персональних асистентів потребує ґрунтовного теоретичного аналізу та оцінки практичних можливостей реалізації в конкретному середовищі виконання. У межах даного розділу було розв'язано кілька взаємопов'язаних завдань, які сформували основу подальшої роботи.

Сучасна наукова література надає широкий спектр визначень, технологій та концепцій розроблення діалогових помічників. Синтез цих джерел дозволив дійти висновків, щодо того, який інструментарій є необхідними для реалізації повноцінного сучасного персонального асистента. Саме ця вимога була покладена в основу критеріїв для подальшого аналізу.

Водночас, було виділено основні тенденції та технічні особливості платформи Signal, що відрізняють його від інших месенджерів. Цей аналіз дозволив визначити обмеження та очікувані властивості системи, які безпосередньо диктують проєктні рішення для будь-яких автоматизованих програмних засобів, що працюють в цьому середовищі.

Оцінка існуючих інструментів з автоматизації, проведена на основі як теоретичних вимог, так і платформних обмежень, дозволила виявити їхні недоліки. Попри наявну практичну потребу, жоден із проаналізованих проєктів не пропонує достатньої якості інструментів для досягнення поставленої мети.

Відповідно, це демонструє, що розробка власного програмного засобу, який би вирішував виявлені проблеми, є актуальною та практично значущою задачею, що усуне виявлений пробіл. Сформульовані у цьому розділі теоретичні засади слугуватимуть безпосередньою основою для проєктування архітектури і розроблення функціоналу такого інструменту.

РОЗДІЛ 2
МЕТОДИ ТА ЗАСОБИ РОЗРОБЛЕННЯ

2.1. Упорядкування вимог до бібліотеки

У попередньому розділі було визначено ключові властивості персональних асистентів і виявлено основні недоліки наявних рішень. Щоб використати отримані результати для подальшого проєктування системи, необхідно упорядкувати їх у вигляді функціональних та нефункціональних вимог для усунення неоднозначності трактування і формування основи для проєктування архітектури.

Функціональні вимоги визначають конкретні операції та завдання, які система повинна бути здатною виконувати. Ці вимоги безпосередньо випливають із раніше розглянутого поняття персональних асистентів та недоліків наявних інструментів. У таблиці 2.1 надано основні характеристики системи, що визначають її очікувану поведінку та можливості.

Таблиця 2.1

Функціональні вимоги до бібліотеки

№ п/п	Назва	Опис
1	Стратегія управління діалогом	Для забезпечення можливості створення повноцінних персональних асистентів бібліотека повинна надавати інструменти для організації потоку діалогу як лінійної або розгалуженої послідовності станів
2	Збереження стану діалогу	Бібліотека має містити інструменти для безпечного збереження контексту та проміжних станів діалогу, оскільки саме на цих даних ґрунтується управління діалогом

Таблиця 2.1 (продовження)

Функціональні вимоги до бібліотеки

№ п/п	Назва	Опис
3	Методи формування відповіді	Бібліотека має надавати зручні та оптимізовані методи зіставлення вхідного повідомлення з шаблоном та вибору відповідного обробника
4	Ізольоване обслуговування користувачів	Бібліотека повинна забезпечувати повну ізоляцію сесій між різними обліковими записами асистентів, користувачами та групами
5	Управління обліковим записом	Бібліотека повинна надавати функціонал управління обліковими записами: реєстрація, верифікація, прив’язка, оновлення, видалення
6	Повнота API	Бібліотека повинна надавати вичерпний набір методів для керування клієнтом, виконання основних операцій (робота з повідомленнями, групами, контактами)
7	Універсальна обробка типів повідомлень	Бібліотека повинна підтримувати обробку різних типів повідомлень (текстових, вкладень, реакцій)
8	Тестованість персональних асистентів	Бібліотека має надавати зручний інструментарій для тестування роботи персональних асистентів без необхідності взаємодії з сервером Signal

Нефункціональні вимоги визначають якісні характеристики системи та випливають як із технічних особливостей платформи Signal, так і з аналізу недоліків існуючих рішень. Ці вимоги упорядковано у таблиці 2.2.

Таблиця 2.2

Нефункціональні вимоги до бібліотеки

№	Назва	Опис
1	Продуктивність та асинхронність	Бібілотечні інструменти не повинні вносити суттєвої затримки у процес обробки повідомлень чи у загальний час відповіді системи
2	Простота використання	Бібліотека повинна надавати інтуїтивно зрозумілий API, що повністю приховує технічну сторону роботи з платформою та зменшує обсяг коду для типових сценаріїв використання
3	Тестування	Вичерпне покриття тестами критично важливих функцій та основних сценаріїв використання
4	Документація та приклади	Бібліотека повинна надавати достатню документацію та приклади використання
5	Відмовостійкість	Бібліотека повинна мінімізувати вплив мережевих або серверних (Signal) збоїв на роботу асистентів

2.2. Обґрунтування вибору засобів реалізації

Вибір технологій і архітектурних рішень визначає майбутню стабільність та гнучкість системи. У цьому пункті розглядаються рішення, прийняті на основі раніше сформульованих вимог. Оскільки розроблювана бібліотека сама виступатиме залежністю в інших застосунках, пріоритетом стало мінімальне використання сторонніх пакетів.

2.2.1. Інтерфейс командного рядка Signal-cli

Для реалізації програмної взаємодії з месенджером Signal на початковому етапі бібліотека покладатиметься на Signal-cli. Розглянуті в попередньому розділі рішення підтверджують придатність до використання програмного забезпечення, побудованого на його основі. Використання цього інструменту

дозволить делегувати всю криптографічну роботу перевіреному рішенню, уникаючи необхідності у написанні власної клієнтської частини, та зокрема надати основу для виконання таких вимог як універсальна обробка типів повідомлень та управління обліковим записом.

Signal-cli працює як повноцінний клієнт Signal на модифікованій версії libsignal-service-java [18], адаптованій з офіційного Android-клієнта, що забезпечує побудову мережевого рівня поверх libsignal [19]. Остання використовується для виконання всіх криптографічних операцій (генерацію ключових пар на Curve25519, протоколи X3DH та Double Ratchet, шифрування AES-256 з HMAC-SHA256), тоді як сама утиліта відповідає за управління життєвим циклом ключів.

Комунікація із серверами Signal відбувається через HTTPS/WebSocket з'єднання, використовуючи libsignal-service-java для формування запитів та підтримки постійного WebSocket каналу для отримання вхідних повідомлень.

Signal-cli надає три варіанти роботи з ним:

- Інтерфейс командного рядка;
- D-Bus інтерфейс для Linux;
- JSON-RPC інтерфейс для взаємодії через сокети, TCP або HTTP.

Цей інструмент має достатній набір функцій як для управління акаунтом в Signal, так і для реалізації функціоналу персонального асистента (рис. 2.1).

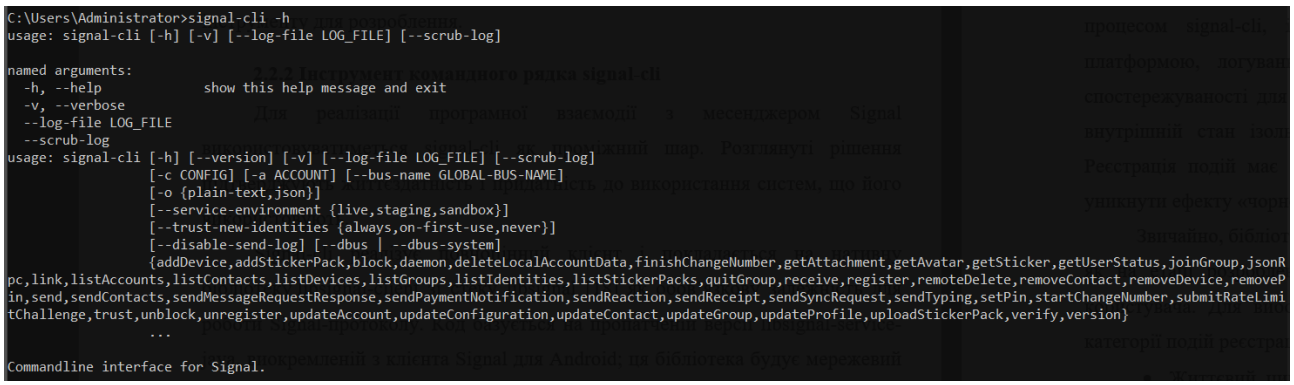


Рисунок 2.1. Довідка Signal-cli, що демонструє перелік доступних параметрів, режимів та команд

В JSON-RPC режимі Signal-cli читає запити зі стандартного вводу або з’єднань TCP/UNIX-сокетів. Вимагається, щоб кожен запит був об'єктом JSON в одному рядку (рис. 2.2).

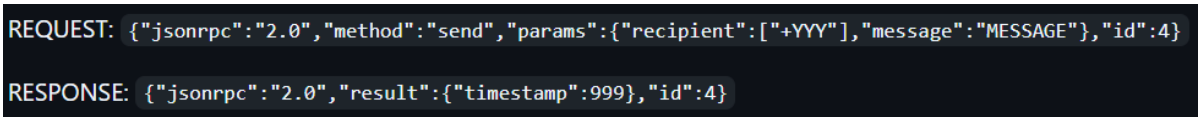


Рисунок 2.2. Приклади JSON-RPC запиту та відповіді signal-cli [10]

Для HTTP надається 3 едпоінти:

- POST /api/v1/rpc: використовується для виклику будь-яких команд, приймає один або батч JSON-RPC запитів;
- GET /api/v1/events: надає Server-Sent Events (SSE) – односторонній постійний потік. Через нього Signal-cli передає вхідні повідомлення, події синхронізації тощо в реальному часі;
- GET /api/v1/check: перевірка стану Signal-cli процесу.

2.2.2. Бібліотека логування Winston

Оскільки бібліотека виступатиме як обгорткою над зовнішнім процесом Signal-cli, так і оркестратором його роботи, логування виступатиме єдиним інструментом забезпечення спостережуваності для користувача бібілотеки, оскільки воно транслюватиме внутрішній стан процесу у доступний для аналізу формат. Реєстрація подій має бути вичерпною та зрозумілою для діагностики, аби уникнути ефекту «чорної скриньки».

Для реалізації підсистеми логування у середовищі Node.js було проаналізовано низку популярних рішень, серед яких Winston, Pino та Bunyan. Будь-який з цих інструментів є достатнім для поставлених завдань, але вибір було зупинено саме на Winston [20], що обґрунтовується наступними можливостями бібліотеки.

Однією з найвагоміших для даної роботи характеристикою Winston є підтримка декількох транспортів, що дозволяє конфігурувати цільові приймачі логів. Користувач отримує можливість самостійно визначати напрямки виводу

інформації: консоль, файлову систему або інтеграцію з хмарними сервісами моніторингу залежно від специфіки впровадження. Winston надає також вбудовану підтримку ротації файлів за розміром чи кількістю.

Також, за допомогою функцій форматування можна впровадити додаткову обробку логів перед записом, наприклад, додавати кастомні поля з метаданими або серіалізувати дані у формат JSON для автоматизованого аналізу чи визначити власний формат, зокрема для маскування даних.

Як і більшість подібних інструментів, Winston надає різні рівні логування з можливістю їх динамічної зміни під час виконання, що з точки зору користувача зручно для діагностики проблем, якщо потрібно тимчасово увімкнути детальніше логування для окремого компонента. Крім того, для кожного рівня можна визначити окремі транспорти.

Водночас, архітектура бібліотеки має дотримуватися принципу інверсії залежностей [21], згідно з яким внутрішні модулі взаємодіють з абстрактним інтерфейсом логера, а не з конкретною реалізацією Winston. Така організація забезпечить можливість заміни логувальної підсистеми за потреби, а також дозволить користувачам інжектувати власні реалізації логерів за умови відповідності визначеному інтерфейсу.

2.2.3. Бібліотека Keyv для сесій

Будь-яка стратегія управління діалогом передбачає, що система поступово накопичує дані від користувача і відповідно реагує на основі цих даних. Зберігання сесій виключно в оперативній пам'яті є достатнім лише для розроблення чи тестування, але не завжди доцільним для продуктового середовища через ризик втрати даних. Також варто враховувати, що бібліотека має залишатись універсальним інструментом, не обмежуючи користувачів у виборі конкретного сховища.

З технічної точки зору об'єкт сесії може бути як невеликою структурою даних, доступ до якої має здійснюватися швидко у процесі обробки вхідних повідомлень, так і містити великий обсяг контекстної інформації залежно від

призначення персонального асистента. Відповідно, час зберігання сесії може сильно варіюватися.

Keyv є бібліотекою для зберігання даних у форматі ключ-значення, яка також забезпечує єдиний інтерфейс (методи set, get, has, delete, clear) для роботи з різними типами сховищ через адаптери для Redis, MongoDB, SQLite, MySQL та багатьох інших БД [22]. У контексті проєкту використання Keyv видається доцільним ще й завдяки таким можливостям:

- Keyv пропонує вбудовану підтримку TTL, що дозволяє делегувати очищення застарілих сесій цій бібліотеці. До того ж, вона обгортає адаптери власною функціональністю управління часом життя навіть для тих баз даних, які не передбачають такого нативно.
- Підтримка просторів імен створює можливість використання одного екземпляру Keyv для збереження сесійних даних від декількох екземплярів ботів, що запобігає конфліктам ключів та дозволяє виконувати очищення даних у межах конкретного іменного простору.
- Забезпечення обробки помилок взаємодії з базою даних, що підвищує загальну стійкість системи.
- Для консистентності даних між різними бекендами бібліотека використовує json-buffer для серіалізації, при цьому залишаючи можливість визначення власних функцій.

Такий підхід звільняє розробника від необхідності власноруч реалізовувати логіку управління контекстом діалогу та гарантує єдиний інтерфейс взаємодії для внутрішніх компонентів бібліотеки без потреби створення власних адаптерів.

2.2.4. Інструмент валідації структур даних Zod

Для того, аби пояснити роль бібліотеки Zod [23] в проєкті і обґрунтувати її вибір, доведеться зробити засвіт щодо того, якою моделлю реалізовано стратегію управління діалогами для персональних асистентів. Якщо коротко: діалог представляється у вигляді графа з вершинами-кроками розмови, а найпростіший

спосіб перевірити коректність його визначення – побудувати конфігураційні об’єкти-схеми для кожного окремого кроку та всього потоку загалом.

Тож, ринок інструментів валідації даних на основі схем представлений багатьма рішеннями, наприклад, AJV, Yup, Zod та Joi. Кожен інструмент має свою область застосування і переваги, зокрема AJV характеризується найбільшою швидкістю виконанння, але обмежується валідацією лише JSON-схем, тоді як Yup та Joi мають більш широкий і зручний API, хоча зорієнтовані переважно на вебформи або мають надмірну складність для завдань бібліотеки.

Тому вибір на користь Zod зумовлений тим, що інструмент підтримує валідацію нетипових для JSON структур, зокрема функцій, колекцій та інших несеріалізовних значень, які є невід’ємною частиною визначення потоків діалогу.

Zod підтримує композицію схем (методи union, intersection та discriminated unions) та дозволяє виконувати перевірки, де коректність одного параметра залежить від значень інших полів або від структури всього конфігураційного об'єкта в цілому (метод superRefine). Також, вона дозволяє визначати значення за замовчуванням для необов’язкових полів, наприклад, версію, опис, чи таймаути, що спрощує визначення конфігурацій для кінцевих користувачів. Така валідація дозволяє перевірити не лише типи примітивних значень, а й виявлити логічні помилки у визначеннях потоків (рис. 2.3).

```
export const flowDefinitionSchema = z
    .object({
        id: flowIdSchema,
        version: flowVersionSchema.default("1.0.0"),
        ttl: ttlSchema.default(null),
        hooks: flowHooksSchema.default({ onEnter: null, onExit: null, onError: null }),
        interrupts: z.array(flowInterruptSchema).default([]),
        nodes: z.map(nonEmptyStringSchema, nodeDefinitionSchema),
        initialNode: nonEmptyStringSchema,
        metadata: metadataSchema.default({}),
        analysis: flowAnalysisSchema,
    })
    .strict()
    .superRefine((value, ctx) => {
        if (!value.nodes || value.nodes.size === 0) {
            ctx.addIssue({
                code: z.ZodIssueCode.custom,
                path: ["nodes"],
                message: `Flow '${value.id}' must contain at least one node`,
            });
        }

        if (!value.initialNode || !value.nodes.has(value.initialNode)) {
            ctx.addIssue({
                code: z.ZodIssueCode.custom,
                path: ["initialNode"],
                message: `Initial node '${value.initialNode}' is not defined in flow '${value.id}'`,
            });
        }
```

Рисунок 2.3. Схема Zod для перевірки фінального визначення сценарію діалогу

Серед недоліків бібліотеки виділяється менша, порівняно з аналогами, швидкість роботи, що насправді не є критичним, оскільки перевірка виконуватиметься одноразово під час запуску бота, і на подальшу продуктивність це не впливатиме.

2.2.5. Фреймворк для тестування Jest

На відміну від кінцевих застосунків, де помилки можуть бути виправлені оперативним оновленням, дефекти у бібліотечному коді впливають на всі проєкти, що її використовують. Це обумовлює важливість комплексного автоматизованого тестування як невід'ємної складової процесу розроблення.

У контексті даного проєкту передбачається виконання трьох стандартних видів тестування: юніт-, інтеграційного та наскрізного. Юніт-тестування має забезпечити верифікацію коректності роботи окремої функціональності бібліотеки в ізольованому середовищі. Інтеграційне тестування, натомість, спрямоване на перевірку взаємодії між компонентами системи, зокрема коректність формування команд та запитів для Signal-cli, обробку відповідей та узгодженість роботи ланцюжків обробників. Наскрізне тестування має підтвердити працездатність всієї бібліотеки.

Внутрішня організація бібліотеки не накладає специфічних обмежень на вибір тестового фреймворку, окрім підтримки ефективного мокування взаємодії зі Signal-cli. Реальна комунікація з месенджером у рамках автоматизованих юніт- та інтеграційних тестів є неприйнятною з кількох причин: це порушує ізольованість тестового середовища, створює залежність від зовнішніх сервісів, уповільнює виконання та може призвести до блокування облікового запису.

Для обґрунтованого вибору проаналізовано декілька популярних та надійних фреймворків, серед яких Jest, Vitest та Mocha. Остаточний вибір на користь Jest [24] обумовлений його зручністю та відповідністю вимогам проєкту. Фреймворк забезпечує нативну підтримку ECMAScript Modules через експериментальні VM-модулі Node.js, надає вбудовані механізми для емуляції зовнішніх залежностей без необхідності реальних звернень до Signal через

методи jest.mock, jest.fn та jest.spyOn, а завдяки хукам beforeEach та afterEach, а також автоматичному скиданню моків, тести виконуватимуться ізольовано, що важливо для коректної перевірки компонентів, які працюватимуть зі спільними даними сесій.

2.3. Загальна архітектура бібліотеки

Архітектура бібліотеки формуватиметься з огляду на те, що взаємодія з Signal на поточному етапі залежатиме від зовнішніх інструментів. Необхідно надати зручний і надійний програмний інтерфейс, який би забезпечив стабільну роботу з месенджером незалежно від використовуваних засобів взаємодії з ним та приховав технічні складнощі від кінцевих користувачів бібліотеки.

Насамперед необхідно визначити загальну концепцію створюваного програмного рішення з точки зору його архітектурної ролі у взаємодії з кодом розробника. Хоча терміни «бібліотека» і «фреймворк» часто використовуються як взаємозамінні, вони представляють принципово різні підходи до організації та способу керування виконанням користувацького коду.

Головна відмінність полягає в понятті інверсії управління. Використання бібліотеки надає розробникові можливість котролювати потік виконання програми і викликати окремі функції бібліотеки у потрібний момент. Натомість, фреймворк сам керує потоком виконання, а розробник лише надає код, який фреймворк викликає у відповідні моменти [25] (рис. 2.4).

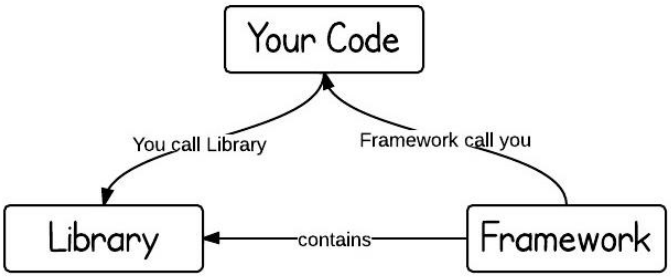


Рисунок 2.4. Розподіл контролю у фреймворковому та бібліотечному підході

Потенційні сценарії використання бібліотеки є досить різноманітними – від ботів-нагадувань з однократною відповіддю до складних діалогових систем з використанням методів штучного інтелекту для підтримки розмови. Тому архітектура бібліотеки має бути достатньо гнучкою, щоб адаптуватися до потреб рішень різної складності.

Для забезпечення такого підходу було обрано мікроядерну аріхтектуру (також відому як плагінну) [26]. Хоча традиційно цей стиль застосовується у складніших системах, його принципи видаються ефективними і для даного проєкту завдяки можливості незалежного розширення функціональності та гнучкого налаштування. Ця архітектура є відносно простою монолітною, що складається з двох основних компонентів: базового ядра системи та плагінних модулів (рис. 2.5).

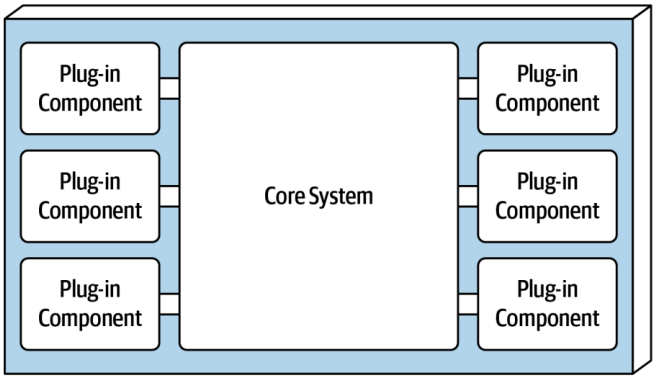


Рисунок 2.5. Базові компоненти плагінної архітектури [26]

Логіка застосунку розподіляється між цими компонентами таким чином, що ядро реалізує лише мінімальний функціонал, необхідний для роботи системи, а вся додаткова функціональність винесена у окремі розширення, що сприяє розширюваності та супроводжуваності, а також спрощує тестування.

У даному проєкті ядром виступатиме обгортка-оркестратор над Signal-cli, що забезпечуватиме перетворення викликів функцій у команди та запити, додатково виконуючи логування, валідацію, обробку повідомлень. Внутрішня структура ядра може бути організована за багаторівневою архітектурою або модульним монолітом [26] (рис. 2.6).

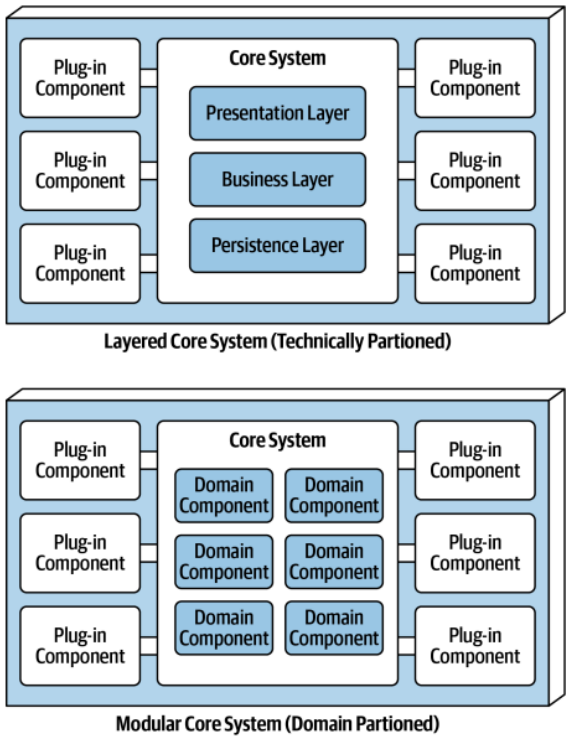


Рисунок 2.6. Варіації для ядра у плагінній архітектурі [26]

Класична тришарова архітектура зазвичай включає рівні представлення, бізнес-логіки та доступу до даних, а оскільки бібліотека не має ні графічного інтерфейсу (принаймні до реалізації візуального конструктора для побудови діалогових сценаріїв), ні взаємодії з базами даних, то вибір зупиняється на звичайному монолітному підході.

У свою чергу, плагіни визначаються, в ідеалі, як незалежні між собою компоненти, які вносять додаткові можливості до системи або ізолюють часто змінюваний код. Комунікація між ними та ядром системи відбувається через виклик методу або функції точки входу класу плагіна. Крім того, плагін може бути підключений до системи динамічно під час рантайму або на етапі запуску

застосунку. Було обрано підхід із підключенням плагінів на етапі компіляції, оскільки це значно простіше в управлінні, хоча й потребує повторного розгортання при їх зміні, додаванні чи видаленні [26].

Застосування цього підходу відкриває можливості підміни конкретних реалізацій розширень, не впливаючи на роботу ядра бібліотеки чи інших плагінів, що дозволяє адаптувати бібліотеку під специфічні вимоги різних проєктів.

Крім технічних переваг, така архітектура дозволяє застосувати гнучку бізнес-модель. Команди та окремі розробники можуть обирати лише необхідну їм функціональність, що може впливати на вартість ліцензії залежно від кількості підключених модулів чи конкретних їх реалізацій. Також, відкривається перспективний напрям розвитку зі створення екосистеми навколо бібліотеки, у якій спільнота зможе додавати власні розширення, зберігаючи основний код компактним, тестованим та надійним.

2.4. Методи та алгоритми реалізації

У цьому пункті розглянуто алгоритмічні рішення та моделі, що лежать в основі програмної реалізації бібліотеки. Описано методи обробки графових структур для розв’язання залежностей розширень та перевірки коректності визначень сценаріїв, застосування теорії скінченних автоматів для впровадження управління діалогом, стратегії оптимізації отримання повідомлень, принципи побудови послідовності проміжних обробників, а також криптографічні алгоритми для захисту даних користувачів.

2.4.1. Визначення порядку ініціалізації розширень

Хоча класичні принципи плагінної архітектури передбачають повну автономність розширень, на практиці це часто не так. Це особливо актуально для відкритої екосистеми, де розробники можуть використовувати наявні плагіни як

основу для власних розширень. Тож, плагін може залежати від одного або декількох інших, які, у свою чергу, можуть мати власні залежності.

Така система утворює структуру, яку можна представити як орієнтований ациклічний граф $G = (V, E)$, де:

V – множина вершин, що відповідає множині плагінів,

E – множина орієнтованих ребер,

$(u, v) \in E$ – відношення залежності між вершинами. Це відношення визначає, що плагін u має залежності від плагіна v і повинен бути ініціалізований лише після завершення ініціалізації v (рис. 2.7).

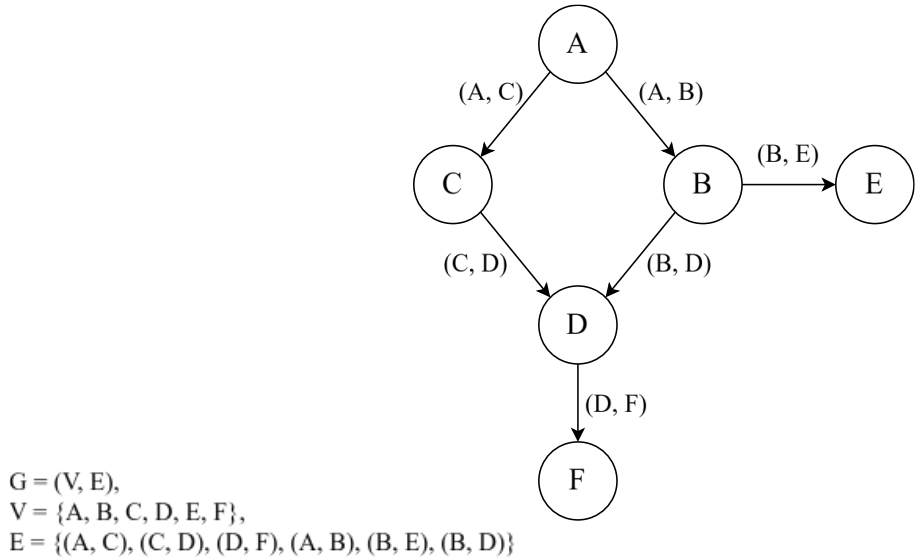


Рисунок 2.7. Залежності плагінів, представлені у вигляді орієнтованого графа

Ациклічність у даному випадку є обов'язковою. Якщо між плагінами утворюється замкнений ланцюг, то неможливо визначити допустимий порядок ініціалізації (рис. 2.8).

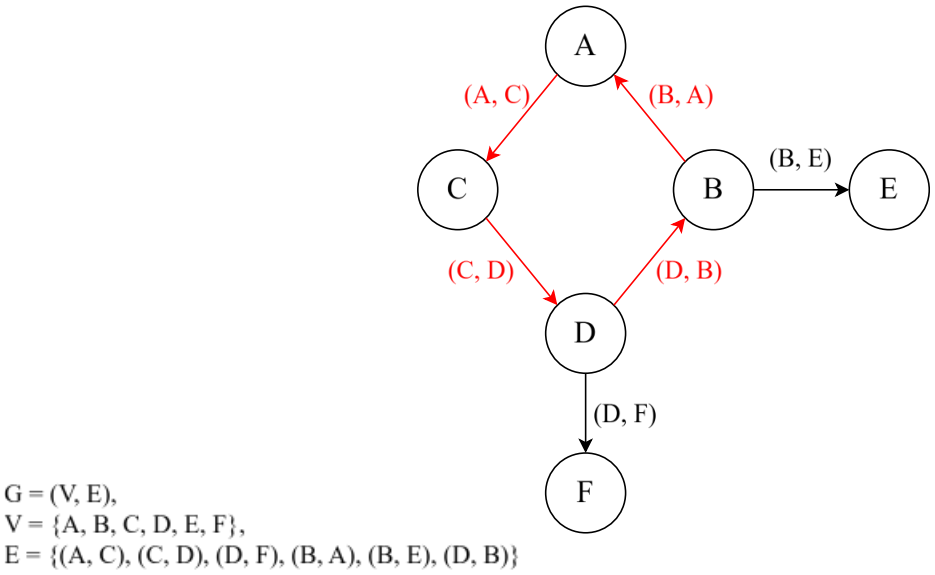


Рисунок 2.8. Орієнтований граф із циклічною залежністю A і B

Для визначення правильної послідовності завантаження використовується алгоритм топологічного сортування на основі методу пошуку в глибину [27].

Для кожної вершини $v \in V$ існує три можливих стани: не відвідана, в процесі обробки та оброблена. Алгоритм рекурсивно обходить граф, встановлюючи для вершини стан «в процесі обробки» при вході в рекурсію та «оброблена» при її завершенні. Виявлення циклів здійснюється через аналіз стану «в процесі обробки»: якщо під час обходу зустрічається вершина, яка вже має цей стан, це означає існування ребра, що повертається до активної вершини, і відповідно формує цикл у графі. Після повної обробки всіх залежностей вершина додається до результуючого списку, що гарантує коректність послідовності ініціалізації плагінів (рис. 2.9).

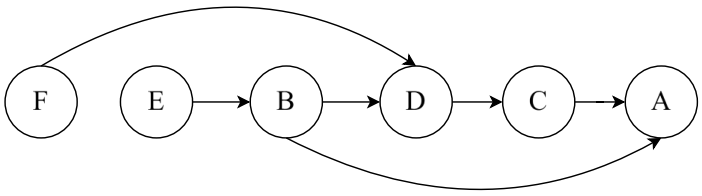


Рисунок 2.9. Один із варіантів топологічно впорядкованої послідовності ініціалізації плагінів з рисунку 2.7

Також, передбачається перевірка повноти графа за умовою

$$\forall (u, v) \in E: v \in V,$$

тобто для кожної залежності має бути присутній відповідний плагін у реєстрі.

2.4.2. Стратегія управління діалогом на основі скінченних станів

Стратегія управління діалогом на основі скінченних станів описується у [28] як набір заздалегідь визначених послідовностей кроків, що представляють стан діалогу у будь-який момент розмови, де кожен такий стан має обмежену кількість можливих переходів до інших станів та визначає набір дій, які діалоговий помічник має виконати в певній ситуації. Зокрема, у цьому дослідженні наводиться приклад на основі FSM (Finite State Machine), що математично описується як $M = (S, I, \delta, s_0, F)$ (рис. 2.10), де

$S = \{s_0, s_1, s_2, \ldots, s_n\}$ – скінченна множина станів,

I – вхідний алфавіт,

$\delta: S \times I \to S$ – функція переходів між станами,

$s_0 \in S$ – початковий стан,

$F \subseteq S$ – множина кінцевих станів.

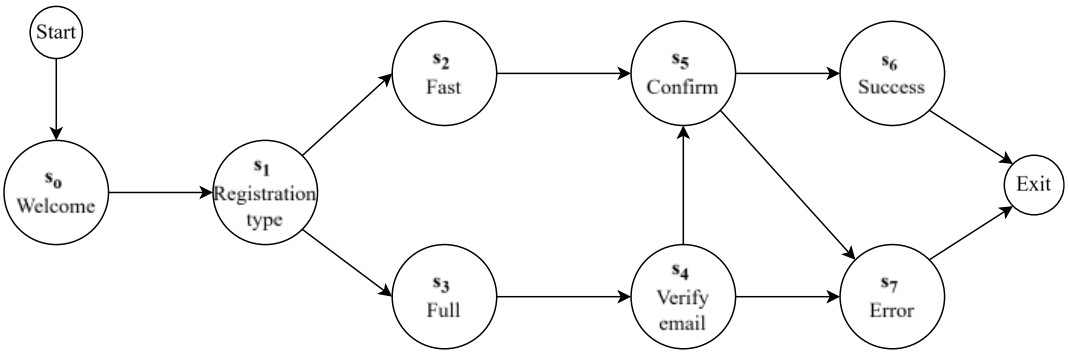


Рисунок 2.10. Представлення діалогу у вигляді графа

Відповідно, множина S – це конкретні етапи розмови з користувачем, I – набір можливих повідомлень чи дій користувача, а функція переходів визначає наступний етап, опираючись на отримані вхідні дані.

Визначення діалогу як графа у контексті сучасного стрімкого розвитку чат-ботів отримав дещо інше застосування, наприклад у [29] вершини вже не описуються лише як етапи розмови, а поділяються на семантичні та діалогові вузли, кожен із яких відповідає за окремі аспекти поведінки системи (рис. 2.11).

Node label	Category
MACRO_INTENT	Knowledge
MICRO_INTENT	Knowledge
SLOT	Knowledge
ENTITY	Knowledge
ENTITY_DICTIONARY	Knowledge
ENTRY	Knowledge
FILLING_FUNCTION	Dialogue
VALIDATION_FUNCTION	Dialogue
ACTION	Dialogue
UTTERANCE	Dialogue
QUESTION	Dialogue

Рисунок 2.11. Структура графу, організована за різними типами вузлів

Відповідно, функція переходів не визначається статично, а виникає як результат семантичного аналізу стану діалогу та наявних даних. Цей підхід потребуватиме детальнішого аналізу для покращення стратегії управління діалогом в майбутньому, але наразі робота зосереджуватиметься на простішому визначенні.

Проте класична FSM має суттєві обмеження при застосуванні до реальних діалогових систем у месенджерах [28]:

1. FSM оперує лише станами, але не даними, тоді як повноцінна комунікація з користовувачем передбачає поступове накопичення і передачу інформації між етапами розмови.
2. Функція переходів δ є детермінованою і не враховує динамічні умови або зовнішній стан системи. Без ускладнення кожної окремої функції переходів неможливо передбачити переривання сценарію у будь-який момент або обробку виключних ситуацій.
3. Автомат змінює стан лише при надходженні вхідного сигналу, часові паузи у взаємодії з користувачем не враховуються.

Перша із зазначених проблем вирішується досить легко – достатньо ввести множину контекстних даних діалогу, які зберігатимуться зовнішньо, що дозволяє діалоговому помічнику накопичувати інформацію, отриману від співрозмовника протягом усієї (або частини) розмови. На основі цих контекстних даних обчислюється нова функція переходів, яка враховує не лише поточний стан та множину вхідних даних, а й попердній контекст розмови.

Для вирішення проблеми детермінованості функції переходів додаються функції переривань, які працюють поверх δ. Тобто, перед обробкою вхідних даних в поточному стані перевірятимуться певні глобальні умови, спрацювання яких переведе сценарій діалогу до іншого етапу розмови або взагалі завершить його виконання.

Оскільки комунікація з персональним асистентом може бути значно розподілена в часі, також додаються функції часових обмежень, що дозволяють реагувати на неактивність користувача. Ці функції визначаються локально на кожному етапі розмови, визначаючи цільовий стан, до якого сценарій діалогу має перейти після завершення часу очікування.

Приклад того, як описаний підхід може працювати на практиці, показано на рисунку 2.12.

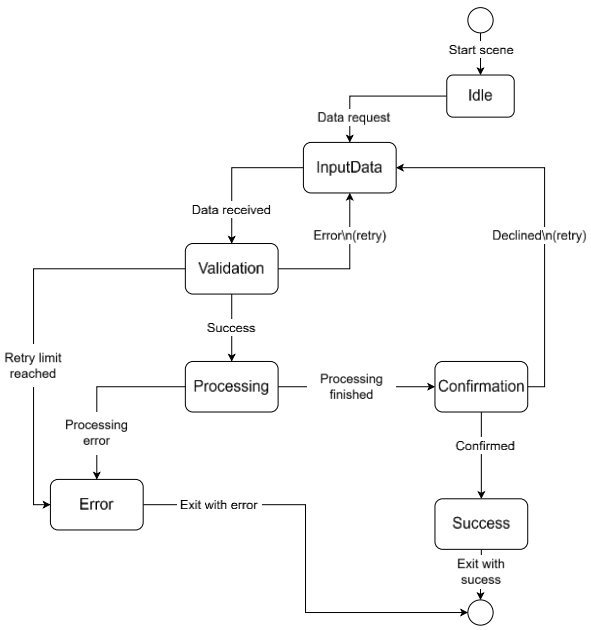


Рисунок 2.12. Графічне представлення потоку діалогу з обмеженою кількістю повторних спроб

2.4.3. Визначення сильнозв'язаних компонент

Складні діалогові сценарії, побудовані на основі FSM, можуть містити циклічні переходи, локальні петлі, розгалуження чи вкладені цикли. Для забезпечення коректності таких сценаріїв необхідно гарантувати можливість їх завершення.

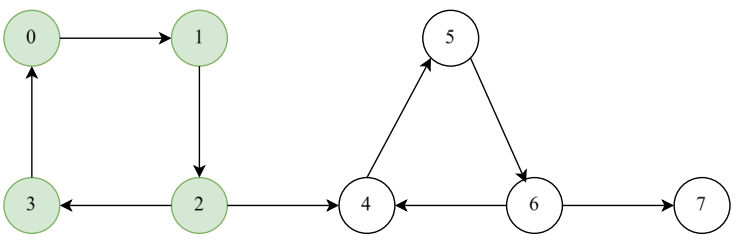
Переходи між вершинами можна представити як ребра орієнтованого графа, де цикли можуть бути як запланованим повторенням кроку, так і помилкою розробника, що ніколи не приводить до вершини завершення. Для виявлення таких проблемних структур застосовується алгоритм Тар'яна для пошуку сильнозв'язаних компонентів.

Сильнозв'язана компонента (SCC) у орієнтованому графі – це множина вершин, кожна з них досяжна з будь-якої іншої.

Алгоритм Тар'яна виконує один прохід DFS по графу, встановлюючи для кожної вершини два значення:

- index – порядковий номер, що присвоюється вершині при першому відвідуванні під час DFS;
- lowlink – найменший індекс вершини, досяжної з поточної вершини, включно через зворотні та бічні ребра.

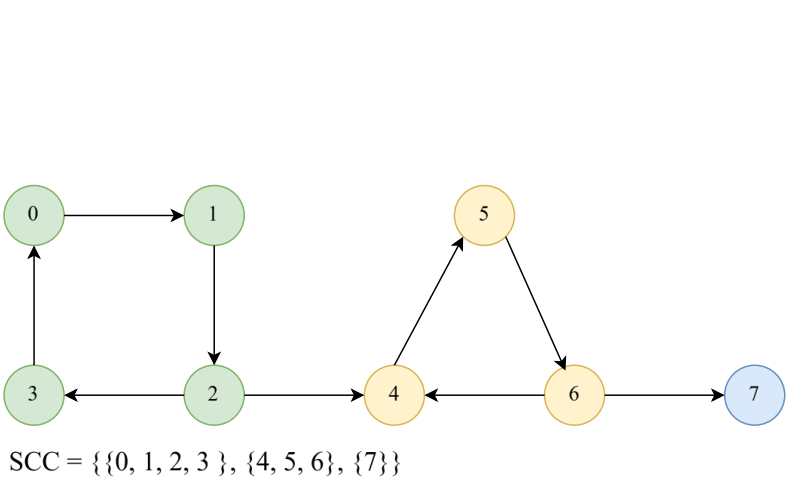
Усі активні вершини під час обходу зберігаються у стеку. Значення lowlink оновлюється на основі аналізу суміжних вершин: якщо сусідня вершина ще не була відвідана, алгоритм спочатку обробляє її рекурсивно (рис. 2.13).



Vertex	Index	Lowlink
0	0	0
1	1	1
2	2	2
3	3	3
4	-	-
5	-	-
6	-	-
7	-	-

Рисунок 2.13. Стан алгоритму після відвідування першої групи вершин, присвоєні значення index та lowlink

Якщо ж суміжна вершина вже присутня у стеку (на рисунку 2.12 3→0), lowlink зменшується до мінімального значення index серед таких сусідів. Коли для певної вершини значення lowlink дорівнює її index, вона визначається як корінь SCC й вилучається зі стеку разом із вершинами до неї, утворюючи одну сильнозв'язану компоненту. Цей процес повторюється для кожної невідвіданої вершини графа, виявляючи всі SCC (рис 2.14).



Vertex	Index	Lowlink
0	0	0
1	1	0
2	2	0
3	3	0
4	4	4
5	5	4
6	6	4
7	7	7

Рисунок 2.14. Результат розбиття графа на сильнозв'язані компоненти

2.4.4. Криптографічні алгоритми для захисту даних сесій

Оскільки бібліотека призначена для роботи з месенджером Signal, який робить акцент на захисті даних, вона повинна дотримуватися стандартів безпеки при роботі з даними користувачів. Для реалізації механізмів безпечного зберігання контексту діалогу було використано криптографічний модуль середовища Node.js crypto, який надає доступ до реалізацій алгоритмів OpenSSL.

Для задач, що потребують отримання фіксованих дайджестів із вхідних даних довільної довжини (таких як ідентифікатори сутностей для ключів сесій), обрано алгоритм SHA-256 (Secure Hash Algorithm – 256-bit). Для бібліотеки важливими є такі його властивості:

- Незалежно від формату вхідних даних (номер телефону, UUID, ідентифікатор групи), результат завжди матиме фіксовану довжину, що

є необхідним для формування уніфікованих ключів пошуку в сховищі даних;

- Незворотність перетворення дозволяє використовувати дайджести замість відкритих даних у структурі ключів для збереження контексту діалогу, приховуючи реальні ідентифікатори;
- Алгоритм забезпечує високу швидкість обчислення, що дозволяє виконувати нормалізацію ключів при кожному зверненні до сесії, що під час роботи персонального асистента траплятиметься постійно.

Для перетворення текстових секретів, що надаються розробником, у криптографічно стійкі ключі шифрування використано PBKDF2 (Password-Based Key Derivation Function 2) з використанням HMAC-SHA256. Додавання солі під час генерації ключа запобігає використанню попередньо обчислених таблиць хешів, а багаторазове повторення хешування ускладнює брутфорс-атаки у випадку використання слабких секретних фраз.

Для забезпечення конфіденційності та перевірки цілісності серіалізованих даних сесій обрано симетричний алгоритм AES з довжиною ключа 256 біт у режимі GCM (Galois/Counter Mode). Цей режим є стандартом для сучасних систем:

- GCM одночасно забезпечує шифрування даних та генерацію тегу автентифікації, що дозволяє автоматично виявляти будь-які спроби модифікації або пошкодження шифротексту в сховищі до моменту його дешифрування;
- Для кожного запису сесії генерується унікальний випадковий вектор ініціалізації (IV). Однакові стани сесій різних користувачів будуть збережені як абсолютно різні шифротексти, що запобігає атакам на основі аналізу патернів даних.

ВИСНОВОК ДО ДРУГОГО РОЗДІЛУ

Обрана сукупність методів, алгоритмів та технологій реалізації формує теоретичну основу для створення надійного та масштабованого засобу для створення персональних асистентів у Signal. Було чітко окреслено межі відповідальності бібліотеки, що, у свою чергу, визначило вибір архітектурного стилю.

Бібліотечна парадигма у поєднанні з плагінною архітектурою гарантує універсальність рішення та відсутність інверсії управління, що забезпечує користувачеві повний контроль. Завдяки цьому бібліотеку можна інтегрувати у широкий спектр застосунків без нав'язування власної моделі виконання.

Обрані засоби реалізації Zod, Keyv, Winston та Jest утворюють надійний інструментарій, на який можна покладатися під час реалізації бібліотеки. У даній роботі, аби не перейматися криптографічниим особливостями та написанням власної клієнтської частини, використовуватиметься утиліта командного рядка Signal-cli як основа для взаємодії з Signal. Однак внутрішня організація бібліотеки дозволить в майбутньому замінити його на власну реалізацію клієнта, без руйнування зовнішнього API та значних змін логіки вищих рівнів.

Вибір підходу на основі FSM для стратегії керування діалогами та запропоновані до нього покращення, доповнені алгоритмами перевірки графів, забезпечують коректність і надійність поведінки асистентів. Водночас, запропонований алгоритм адаптивного опитування з експоненційною затримкою вирішує задачу оптимізації системних ресурсів, забезпечуючи компроміс між швидкістю реакції бота та навантаженням на систему.

Використання криптографічних примітивів для хешування, генерації ключів і шифрування даних сесій гарантує виконання вимог до конфіденційності й цілісності інформації, що робить бібліотеку безпечною для реального використання у середовищі з високими вимогами до приватності.

РОЗДІЛ 3
РОЗРОБЛЕННЯ І ТЕСТУВАННЯ БІБЛІОТЕКИ ТА ЇЇ РОЗШИРЕНЬ

У цьому розділі розглядається втілення проєктних рішень у спеціалізовану бібліотеку для створення персональних асистентів у месенджері Signal. До уваги пропонується опис технічної реалізації основи бібліотеки, а також огляд набору розширень, що надають найважливіший інструментарій. Розділ завершується оцінюванням коректності роботи застосованих алгоритмів на основі їх практичного випробування та автоматизованого тестування.

3.1. Структура проєкту

За архітектурою плагінів передбачається, що ядро визначає загальний контракт плагіна (ідентифікатор, функції керуванн, залежності) і точки розширення, а конкретний функціонал додається через окремі плагіни, які підключаються та відключаються без зміни ядра (рис. 3.1).

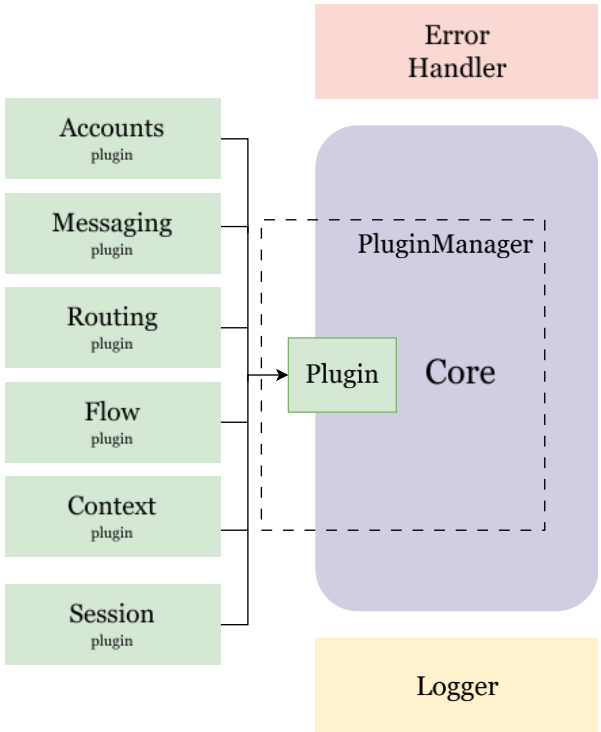


Рисунок 3.1. Загальна структура бібліотеки на основі плагінного підходу

У проєкті контракт плагінів визначає клас Plugin через набір обов'язкових властивостей: унікального імені, методів ініціалізації, реєстрації, видалення розширень, а також setLogger для отримання екземпляру логера від ядра та extendClient – кожен ключ, що повертається ним, приєднується як метод чи поле до екземпляра клієнта, розширюючи API без змін ядра (рис. 3.2).

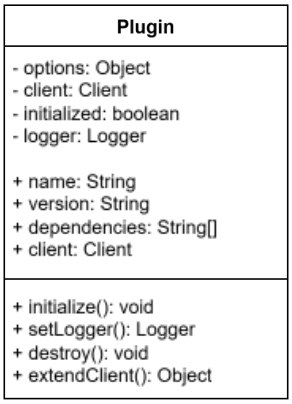


Рисунок 3.2. Діаграма класу Plugin

Опціонально для плагінів можна вказати масив залежностей від інших плагінів та версію.

Клас PluginManager є централізованим реєстром плагінів та координатором їх життєвого циклу. В ньому також виконується основна логіка перевірки відповідності плагінів контракту: унікальність імені, наявність обов’язкових методів.

Оскільки користувач бібліотеки може використовувати довільну кількість розширень у довільній комбінації, застосовується алгоритм топологічного сортування для визначення коректного порядку ініціалізації. Виконується рекурсивний обхід графа залежностей плагінів через пошук у глибину (рис 3.3).

```
const visit = (pluginName) => {
    if (visited.has(pluginName)) return;

    if (visiting.has(pluginName)) {
        return new Error(`Circular dependency detected for plugin '${pluginName}`);
    }

    const plugin = this._plugins.get(pluginName);
     visiting.add(pluginName);

    const dependencies = plugin.dependencies || [];
    for (const depName of dependencies) {
        if (!this._plugins.has(depName)) {
            return new Error(`Plugin '${pluginName}' depends on '${depName}' which is not registered`);
        }
        visit(depName);
    }

    visiting.delete(pluginName);
    visited.add(pluginName);
    order.push(pluginName);
};
```

Рисунок 3.3. Фрагмент коду реалізації алгоритму топологічного сортування залежностей плагінів

Алгоритм використовує три змінні:

- visited – повністю оброблені вузли,
- visiting – вузли, що перебувають у стеку рекурсії,
- order – побудована топологічна послідовність, яка визначає порядок запуску плагінів.

При виявленні циклічної залежності, коли вузол знаходиться в visiting, або відсутності залежного плагіна, ініціалізація системи завершується з помилкою.

У межах даної роботи розроблено набір стандартних розширень, що покривають основні потреби при створенні персональних асистентів. Їх перелік подано у таблиці 3.1.

Таблиця 3.1

Розширення бібліотеки

Розширення	Призначення та функції
Accounts	Управління множинними обліковими записами Signal, моніторинг файлової системи для автоматичного виявлення нових акаунтів
Messaging	Отримання вхідних даних через цикли опитування або відкриття потокового з’єднання з Signal-cli

Таблиця 3.1 (продовження)

Розширення бібліотеки

Розширення	Призначення та функції
Context	Створення та управління об'єктом контексту, що інкапсулює дані окремих повідомлень та надає методи для швидкої відповіді
Flow	Реалізує рушій для виконання діалогових сценаріїв на основі скінченних автоматів. Забезпечує валідацію цілісності графа переходів та управління виконанням стратегії діалогу
Session	Управління контекстом діалогів із алгоритмами захисту даних
Routing	Маршрутизація повідомлень із застосуванням предикатів і фільтрів та формуванням послідовності проміжних обробників

Основний компонент бібліотеки – ядро – організовано як модульний моноліт з декількома внутрішніми шарами. Для зручності використання розробником функціональність Signal-cli було розділено на окремі домени.

Account: відповідає за функції реєстрації та верифікації нового облікового запису, його видалення з очищенням локальних даних, встановлення або видалення пін-коду, оновлення технічних налаштувань облікового запису (наприклад, налаштування політик щодо видимості номера телефону для інших користувачів, встановлення юзернейму) та додаткових даних (імені, аватару, опису).

Оскільки реалізований Signal-cli клієнт не є офіційним, досить часто при реєстрації вимагається проходження капчі. Для автоматизації цього процесу було реалізовано допоміжну утиліту, яка перехоплює помилку капчі та аналізує повідомлення для визначення конкретного типу (перевищення ліміту запитів чи

власне реєстрації) та встановлює для неї відповідний код. При отриманні токену від розробника спроба реєстрації повторюється з наданим токеном.

Client: тут надаються методи для конфігурації технічних аспектів клієнта, прив'язки нового пристрою до облікового запису (якщо клієнт Signal-cli використовується як основний пристрій) та навпаки – доєднання вже зареєстрованого облікового запису до клієнта, а також синхронізації даних між пристроями.

У Contact, відповідно, підтримується функціональність управління контактами: додавання, оновлення, видалення, блокування чи розблокування, а також встановлення політик довіри до користувачів та підтвердження safety number, тобто перевірки цілісності криптографічних ключів контакту.

Функціональність надана доменом Group є досить обмеженою і безперечно потребує доопрацювання в майбутньому: доєднатися або вийти з групи, переглянути список активних груп чи видалити локальні дані про них.

Нарешті, у Message надаються методи для отримання повідомлень з додатковою фільтрацією (лише з груп чи приватних чатів, лише текстові чи із вкладеннями тощо), надсилання повідомлень до різних типів діалогів чи у нотатки, а також імітації набору тексту перед відправкою.

Основний компонент – клас SignalClient, який виступає фасадом всього рішення.

Тож, у бібліотеці реалізований достатній набір функцій для роботи персонального асистента та його попереднього налаштування, усуваючи необхідність безпосередньої взаємодії розробника з утилітою Signal-cli.

3.2. Розроблення ядра бібліотеки

Утиліта signal-cli може бути використана як CLI-інструмент, фоновий процес через протокол JSON-RPC або D-Bus-служба. Розробку було зосереджено на перших двох варіантах, оскільки D-Bus орієнтований переважно на Linux, а в Signal-cli ще й має низку відкритих проблем [6]. Натомість

передбачено підхід на основі Docker-образів Signal-cli, що активно підтримуються спільнотою.

Для реалізації цих способів виконання Signal-cli застосовується підхід на основі патерну "Стратегія" через абстрактний клас Executor та три конкретні виконавці: NativeExecutor, DockerExecutor та JSONRPCExecutor.

У класі Executor визначено абстрактні методи, які реалізуються конкретними стратегіями відповідно до середовища виконання (рис. 3.4).

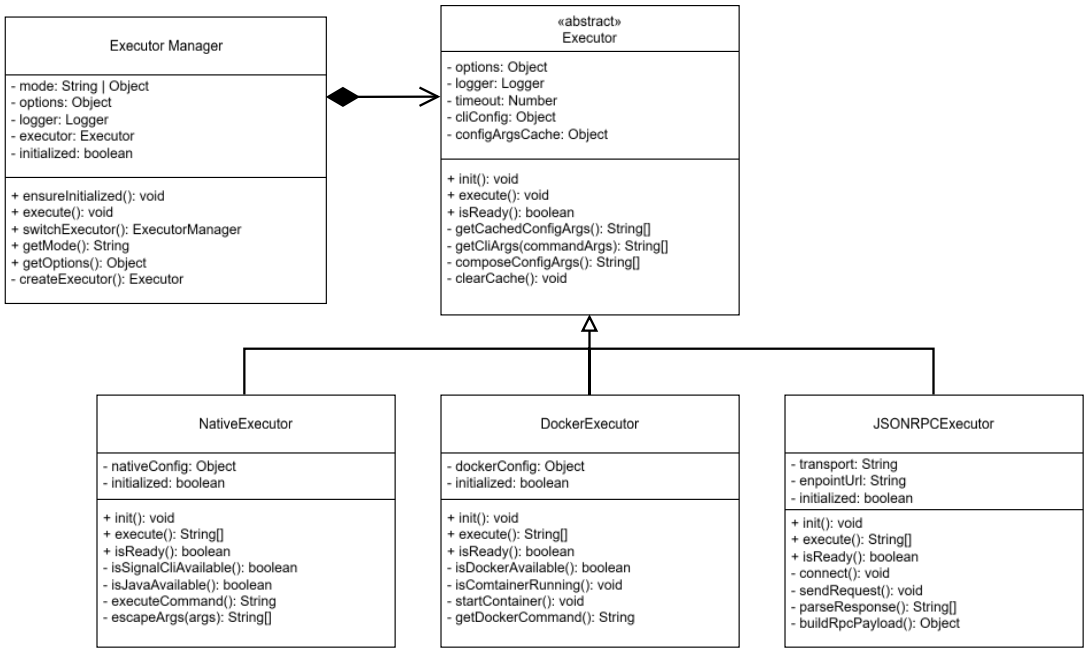


Рисунок 3.4. Ієрархія класів виконавців команд

Клас ExecutorManager є контекстом у цьому патерні, який обирає конкретну стратегію відповідно до заданого режиму та ініціалізує її. Це забезпечує можливість розширювати набір стратегій і підмінювати їх тестовими реалізаціями. Метод ensureInitialized перевіряє готовність поточного виконавця, execute делегує виконання команди обраній стратегії. Метод switchExecutor дозволяє змінити стратегію в процесі роботи з очищенням кешу аргументів попереднього екземпляру.

Конкретні виконавці реалізують власну логіку роботи з Signal-cli: запуск локального процесу через spawn (NativeExecutor), виконання через docker exec

(DockerExecutor), надсилання JSON-RPC запитів на порт (JSONRPCExecutor). Спільним для них залишається обробка виводу та помилок.

За будь-якого способу використання Signal-cli на бібліотеку покладається завдання формування команд і запитів для роботи з ним, адже навіть його HTTP-інтерфейс обмежено двома ендпоінтами: /rpc для надсилання запитів і /events для отримання подій через SSE.

Структура CLI-команд та JSON-RPC-запитів є практично ідентичною: використовуються однакові назви методів та параметри. Тому для їх формування застосовується підхід на основі однакових схем. Всі інструкції описано у вигляді JavaScript-об'єктів та згруповано за функціональними категоріями (account, message, group, contact, client).

Кожна така схема визначає перелік методів, набір допустимих опцій та відповідних прапорців, для яких визначено внутрішні категорії (текст, ідентифікатор, посилання, токен, булеве значення тощо) для подальшої обробки, а також технчні параметри: таймаут, необхідність очікування відповіді та спосіб її парсингу (рис. 3.5).

```
[COMMANDS.QUIT_GROUP]: {
    timeout: TIMEOUTS.DEFAULT,
    needsResponse: false,
    description: " ",
    category: "group",
    options: {
        groupID: {
            flag: OPTIONS.GROUP_ID,
            type: DATA_TYPES.IDENTIFIER,
            description: " ",
        },
        deleteData: {
            flag: FLAGS.DELETE,
            type: DATA_TYPES.BOOLEAN_FLAG,
            description: " ",
        },
    },
},
```

Рисунок 3.5. Визначення інструкції «Залишити групу»

На основі цих визначень за допомогою утиліти RequestAssembler формуються команди та запити, до яких за потреби додається кеш аргументів з класу Executor (рис. 3.6).

```
args: [
    '--output', 'json'
    '--service-environment' 'live',
    '--timeout', '3',
    '--account', '+380...',
    'quitGroup',
    '--group-id', 'dGV...',
    '--delete'
],
```

```
req = {
    'jsonrpc': '2.0',
    'method': 'quitGroup',
    'params': {
      'account': '+380...',
      'groupId': 'dGV...',
      'delete': 'true'
    },
    'id': '1764264003'
},
```

Рисунок 3.6. Приклад команди та тіла запиту «Залишити групу»

Цей підхід не характеризується найвищою швидкістю виконання порівняно з умовними конструкціями, які використовувались на початку життя розробки, проте така структура опису інструкцій є наочною та легко інтерпретуватиметься іншими розробниками, дозволяє автоматично генерувати документацію на основі полів description та створює основу для потенційної заміни Signal-cli.

Перетворення викликів API бібліотеки у формат, зрозумілий для Signal-cli, виконується на рівні бізнес-логіки ядра. Клас CommandDispatcher для формування об’єктів звертається до утиліти формування команд і делегує її виконання до ExecutorManager.

Адаптери узгоджують API бібліотеки з командним інтерфейсом Signal-cli, виконуючи такі кроки: валідація вхідних параметрів, опціональна попередня або пост-обробка виклику, делегування виконання до відповідного методу диспетчера. Оскільки ця послідовність виконується для більшості команд, і помилки можуть виникнути на будь-якому з цих етапів, її організовано як ряд обробників за шаблоном «Конвеєр».

Усі етапи обробки визначені у публічному методі батьківського класу, тому в окремих методах адаптерів лишається лише опис конфігурації (рис. 3.7).

```
async quitGroup(account, groupID, { clearData = false } = {}) {
    const options = {
        ...(clearData && { deleteData: true }),
    };

    return await this.invoke(
        DISPATCHER_METHODS.QUIT_GROUP,
        {
            validations: [() => validator.isGroupID(groupID)],
            options,
            errorContext: clearData
                ? "Failed to leave group and delete local data"
                : "Failed to leave group",
        },
        account,
        groupID
    );
}
```

Рисунок 3.7. Приклад типового методу адаптера – «Залишити групу»

Валідація параметрів виконується на двох рівнях: логічна валідація бізнес-правил, наприклад перевірка відсутності взаємовиключних комбінацій параметрів, та технічна валідація форматів даних, зокрема відповідність номера телефону стандарту E.164 або валідність ідентифікатора групи у форматі Base64.

Повну послідовність взаємодії компонентів бібліотеки під час виконання виклику «Залишити групу» зображено на рис. 3.8.

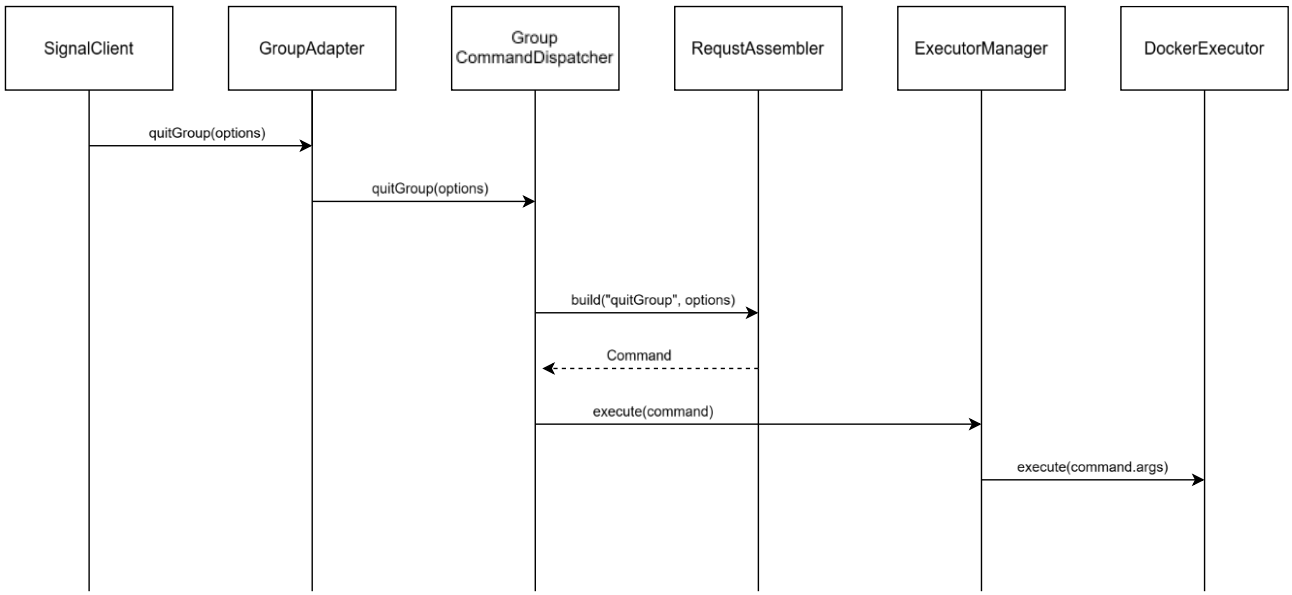


Рисунок 3.8. Діаграма послідовності для операції «Залишити групу»

3.3. Робота з повідомленнями

В основі роботи персонального асистента лежить цикл взаємодії з користувачем, що передбачає прийом вхідної інформації та її послідовну обробку. Наступний виклад деталізує шлях проходження повідомлень через внутрішню логіку бібліотеки: від отримання, крізь послідовні перетворення, до накопичення контексту діалогу та його збереження.

3.3.1. Стратегії отримання повідомлень

Функціональність для безперервного отримання подій від Signal реалізована як окреме розширення Messaging. Головний компонент побудовано на основі подієвої моделі, що є стандартним підходом для подібних до цієї бібліотек. Він координує процес та генерує події надходження нових пакетів даних, які передаються для подальшої обробки персональним асистентом.

Передбачено дві стратегії отримання повідомлень. Перша – polling – реалізує циклічне опитування серверів через команду Signal-cli у режимі інтерфейсу командного рядка. Друга використовує підписку на потік подій – stream – з використанням протоколу JSON-RPC.

Особливістю реалізації полінгу є впровадження алгоритму адаптивного опитування, який динамічно регулює інтервали між запитами залежно від інтенсивності трафіку (рис. 3.9).

```
getPollingDelay(messageCount, idleDuration) {
    if (messageCount > 0) {
        return this.activeDelay;
    }

    if (idleDuration <= this.idleThreshold) {
        return this.idleDelay;
    }

    const interval = this.idleThreshold || 1;
    const time = Math.max(0, idleDuration - this.idleThreshold);
    const koef = Math.min(1, time / interval);

    return this.idleDelay + koef * (this.longIdleDelay  - this.idleDelay)
}
```

Рисунок 3.9. Реалізація алгоритму з лінійною інтерполяцією

Логіка методу ґрунтується на відстеженні часу останньої активності idleDuration: мінімальна затримка activeDelay застосовується під час активного надходження повідомлень, за їх відсутності використовується стандартна затримка idleDelay, доки тривалість часу бездіяльності не перевищить встановлений поріг idleThreshold. Після перевищення цього порогу відбувається плавний перехід до значення максимальної затримки longIdleDelay за формулою лінійної інтерполяції.

У цикл отримання повідомлень також інтегровано обчислення експоненційної затримки у разі виникнення певної послідовної кількості помилок (рис. 3.10).

```
getErrorRetryDelay(attempt) {
    if (!this.useExponentialBackoff) {
        return this.errorRetryDelay;
    }

    return Math.min(
        this.maxBackoffDelay,
        this.errorRetryDelay * Math.pow(this.backoffMultiplier, attempt - 1)
    );
}
```

Рисунок 3.10. Визначення затримки між повторними спробами отримання повідомлень

Лічильник послідовних помилок збільшується при кожній невдалій спробі опитування та скидається при успішному отриманні відповіді. Після перевищення порогу максимальної кількості помилок спроби припиняються і генерується подія помилки.

Всі параметри алгоритмів адаптивного опитування та експоненційної затримки, а також те, чи використовуються вони взагалі, задається розробником у конфігураційному об’єкті класу.

3.3.2. Маршрутизація та обробка повідомлень

Signal надсилає різнотипний потік подій, що містить як змістовні повідомлення користувачів, так і службову інформацію, яка зазвичай не має

практичної цінності для діалогових помічників, до того ж ускладнювала розроблення бібліотеки. З цієї причини на рівні ядра було реалізовано первинну обробку вхідних даних, що дозволяє зменшити обсяг нерелевантної інформації, яка потрапляє до основної логіки асистента.

Розробник має можливість задати конфігурацію для фільтрації даних за типом: змістовні повідомлення (текстові дані або вкладення), індикатори набору тексту, підтвердження доставки і прочитання повідомлення, події синхронізації, інформація про дзвінки, реакції тощо. Звузити цю вибірку можна за допомогою додаткових методів: обробляти лише текстові повідомлення чи події виключно з групових або приватних чатів.

Після фільтрації виконується нормалізація даних, метою якої є їх приведення до єдиної структури, зручної для подальшого використання у розширеннях бібліотеки (рис. 3.11).


```
{
  id: 'f7ce38e4-b484-4027-b37a-b5e6ac937327_1764373083272',
  account: '+49         21',
  sender: {
    uuid: 'f7ce38e4-b484-4027-b37a-b5e6ac937327',
    number: '+38        18',
    name: 'shiw u sa',
    device: 1
  },
  timestamp: 1764373083272,
  serverTimestamp: 1764373083796,
  deliveredTimestamp: 1764373090073,
  type: 'text',
  isGroup: true,
  text: 'aboba',
  expiresIn: 0,
  viewOnce: false,
  group: {
    id: 'KHDw39so4NmGDF2nSs8Tcf2Gh40UsQgjVVBIzlbD4Bs=',
    name: 'new group',
    revision: 2,
    type: 'DELIVER'
  }
}
```


Рисунок 3.11. Об’єкт нормалізованого повідомлення

Необхідність вручну перевіряти контекст діалогу та формувати параметри адресації для кожного виклику зумовила потребу у розробці розширення Context.

Контекст зберігає основні поля повідомлення (рис. 3.12), а також надає методи для автоматичного визначення адресата, який може бути групою або

окремим користувачем. Після визначення отримувача ці методи здійснюють композицію аргументів та викликають відповідний метод для надсилання повідомлення з адаптера Message.

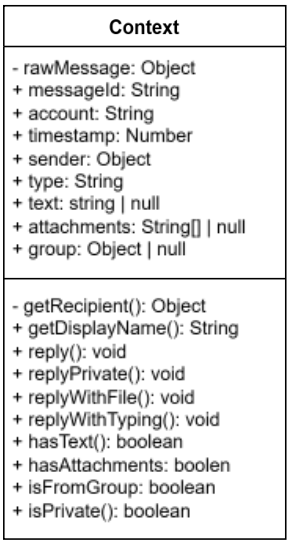


Рисунок 3.12. Діаграма класу Context

Метод reply є універсальним способом надсилання відповіді, тоді як replyPrivate, replyWithFile і replyWithTyping формують власні конфігурації виклику та передають їх reply. Метод replyWithTyping, зокрема, попередньо імітує процес набору тексту.

Для зручності та спрощення логічних перевірок у коді персонального асистента також було реалізовано допоміжні утиліти getDisplayName, hasText, hasAttachments, isFromGroup та isPrivate.

Розширення бібліотеки підключають власні етапи обробки контексту повідомлення, тому для забезпечення їх узгодженої роботи було реалізовано клас Composer, який формує впорядкований ланцюг проміжних і кінцевих обробників та керує їх послідовним виконанням.

Проміжні обробники у класі представлені масивом middleware, кінцеві – handlers, а побудова єдиного ланцюга здійснюється методом compose, який формує кешовану функцію composedFn та використовує лічильник індексу для

контролю порядку викликів, що обмежує повторний виклик next у межах одного обробника (рис. 3.13).

```
const allMiddleware = [...this._middleware];

if (this._handlers.length > 0) {
    allMiddleware.push(async (ctx) => {
        for (const handler of this._handlers) {
            await handler(ctx);
        }
    });
}

if (allMiddleware.length === 0) {
    this._composedFn = async () => {};
    return this._composedFn;
}

this._composedFn = async (ctx) => {
    let index = -1;

    const runner = async (i) => {
        if (i <= index) {
            throw new Error("next() called multiple times in same middleware");
        }
        index = i;

        if (i >= allMiddleware.length) {
            return;
        }

        const middleware = allMiddleware[i];
        return await middleware(ctx, () => runner(i + 1));
    };

    return runner(0);
};
return this._composedFn;
```

Рисунок 3.13. Фрагмент коду формування ланцюга обробників

Композиція обробників виконується статичним методом compose, який об’єднує послідовності middleware та handler кількох екземплярів Composer у спільний ланцюг шляхом об’єднання їх масивів у заданій послідовності (рис. 3.14).

```
static compose(...composers) {
    const merged = new Composer();

    for (const composer of composers) {
        if (!(composer instanceof Composer)) {
            throw new Error("Can only compose Composer instances");
        }
        merged._middleware.push(...composer._middleware);
        merged._handlers.push(...composer._handlers);
    }

    return merged;
}
```

Рисунок 3.14. Композиція обробників кількох Composer у єдиний екземпляр

Окрім стандартних методів on та use, які додають обробник до відповідного масиву, до Composer було вирішено додати декілька алгоритмічних розширень ланцюжка: метод filter перевіряє умову і передає виклик далі у разі її виконання, інакше викликає альтернативний обробник; optional дозволяє ігнорувати помилки і завжди викликати next; fork повертає копію (shallow copy) екземпляра компонувальника; lazy дозволяє відкласти створення до моменту безпосередньої обробки.

Якщо Composer відповідає за технічну організацію послідовного виконання ланцюжка обробників, то Router додає семантичну логіку умовного розгалуження, дозволяючи визначати, які саме обробники мають бути активовані для конкретного вхідного повідомлення.

Визначення маршрутів повідомлень полягає у формуванні ланцюгів обробки, що складаються з набору умов та кінцевого обробника (рис. 3.15).

```
router
    .when((ctx) => !hasAnyActiveFlow(ctx))
    .command("status")
    .and(isGuest)
    .useLocal(makeStatusSnapshot("guest console"))
    .do(async (ctx) => {
        const name = ctx.getDisplayName();
        const status = getSessionStatus(ctx);
        await ctx.reply(
            `${ctx.statusSnapshot.label.toUpperCase()} - ${name}\n` +
                `- Status: ${status}\n` +
                `- Messages seen: ${ctx.statusSnapshot.messages}\n` +
                "You can /changestatus user to unlock detailed info."
        );
    });
```

Рисунок 3.15. Приклад структури обробника команди

Призначенням методів-умов when, and, or та not є фільтрація контексту ще до аналізу змісту повідомлення, що розділяє логіку маршрутизації та бізнес-логіку персонального асистента. За допомогою useLocal можна додати локальний проміжний обробник, який виконуватиметься лише в межах конкретного маршруту. Ці методи можуть бути визначені в будь-якому порядку або не використовуватись взагалі.

Однак обов’язковим у визначенні є метод do, який додає до масиву handlers функцію-обробник, та методи для співставлення вхідних даних. Нижче наведено їх перелік у порядку пріоритету виконання:

1. Команди (command) перевіряють вхідні дані на наявність префікса команди, шаблону самої команди або її псевдоніму. Префікс команди за замовчуванням встановлено як символ «/», але він може бути перевизначений у конфігурації (рис 3.16).

```
const router = client.createRouter({
    commandPrefix: "!",
    commandAliases: {
        "info": ["get", "about"],
    }
});

// ctx.text: "!about amd ryzen 7800x3d"
router
    .command("info")
    .do(async (ctx) => {
        const args = ctx.match.args; // ["amd", "ryzen", "7800x3d"]
        const query = args.join(" ");
        await ctx.reply(`Searching for: ${query}`);
    });

const composer = router.compile();
```

Рисунок 3.16. Приклад визначення команди з псевдонімами та обробкою аргументів

2. Текстові шаблони (hears) предбачають аналіз тексту повідомлення за ключовими словами або регулярним виразом (рис 3.17).

```
router
    .hears(/^(how much|price)/i)
    .do(async (ctx) => {
        await ctx.reply("It is for free.");
    });

router
    .hears(["hi", "hello"])
    .do(async (ctx) => {
        const name = ctx.getDisplayName();
        await ctx.reply(`Hi there ${name}!`);
    });
```

Рисунок 3.17. Співставлення тексту вхідного повідомлення

3. Для подій (on) виконується пряма перевірка типу повідомлення, який було встановлено ще на етапі нормалізації повідомлення (рис. 3.18).

```
router
    .on("attachment")
    .do(async (ctx) => {
        const attachment = ctx.attachments[0];
        await ctx.replyWithFile(
            attachment.localPath,
            "Is it really something worth my attention?"
        );
    });
```

Рисунок 3.18. Обробка повідомлення з вкладенням

Незважаючи на гнучкість такого підходу до обробки повідомлень, у разі необхідності взаємодії з користувачем на основі *діалогового* контексту він швидко переросте у громіздку структуру вкладених перевірок.

3.3.3. Структура діалогових потоків

Щоб усунути накопичення умовних конструкцій і забезпечити однозначність реакції персонального асистента залежно від поточного стану взаємодії з користувачем, було розроблено розширення FlowPlugin, яке дозволяє змоделювати діалог як послідовність кроків. Описану раніше ідею з розширенням FSM зображено на рисунку 3.19.

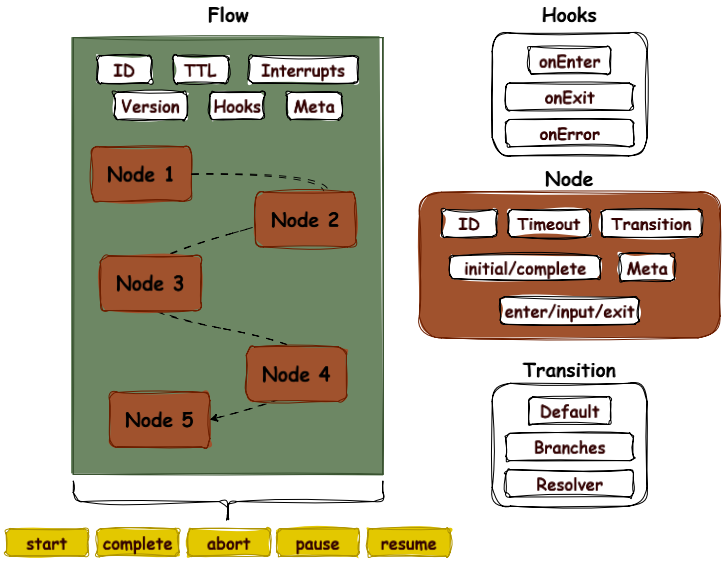


Рисунок 3.19. Концепція діалогу у FlowPlugin

У межах цього розширення діалог описується як один або декілька окремих «потоків», кожен з яких має власну структуру та набори правил. Потік визначає перелік доступних станів і допустимі переходи між ними, формуючи орієнтований граф, у якому вузли (nodes) становлять кроки взаємодії, а ребра – переходи між вузлами (transitions).

Кожен вузол потоку – це окремий об’єкт, що визначає логіку та налаштування для обробки окремого кроку діалогу, завдяки чому їх можна ізольовано тестувати та повторно використовувати у різних сценаріях. Технічно вузол ідентифікується рядковим літералом, який має бути унікальним у межах одного Flow, і описує бізнес-логіку кроку та правила переходів (рис. 3.20), безпосереднє виконання яких виконує машина станів:

1. Default – це лінійний перехід до явно вказаного вузла, який відбувається автоматично після завершення поточного кроку, якщо всі інші типи переходів не визначені або якщо їх умови не спрацювали.
2. Branches – це впорядкований масив об'єктів, у якому задаються умови when, які перевіряється послідовно до тих пір, поки одна з них не поверне істинне значення. Після знаходження збіжної умови обирається відповідний їй цільовий вузол to, а всі наступні умови ігноруються.
3. Resolver – це функція, яка викликається у момент виконання і, за правдивості визначених у ній умов, може повернути ідентифікатор цільового вузла. Виконання переходу до такого вузла матиме найвищий пріоритет для машини станів.

```
.node("node_process_command", (node) =>
    node
        .enter(async (ctx) => {
            await ctx.reply("Enter a command or type 'help'");
            ctx.session.lastPrompt = Date.now();
        })
        .onInput(async (ctx) => {
            ctx.session.lastCommand = ctx.text.toLowerCase().trim();
            ctx.session.commandCount = (ctx.session.commandCount || 0) + 1;
        })
        .resolve(async (ctx) => {
            if (maintenance) return "node_maintenance";
            if (ctx.session.banned) return "node_limit_access";
            return null;
        })
        .branch([
            { when: async (ctx) => ctx.session.isAdmin === true, to: "node_full_access" },
            { when: async (ctx) => /^(help|\?)$/i.test(ctx.message.text), to: "node_show_help" },
            { when: async (ctx) => ctx.session.commandCount > 10, to: "node_limit_access" },
        ])
        .next("node_execute_command")
)
```

Рисунок 3.20. Вузол потоку з визначенням умовних, динамічних та лінійних переходів

Перехід також можна визначити за допомогою timeout, у якому необхідно вказати час очікування, опціонально функцію-обробник таймауту та ідентифікатор цільового вузла (рис. 3.21).

```
.timeout(
    20 * 1000,
    "finish",
    async ({ ctx }) => {
        await ctx.reply(
            "Timeout reached. Moving to finish node."
        );
    }
)
```

Рисунок 3.21. Фрагмент коду вузла з визначенням переходу після завершення часу очікування

Аби уникнути перевантаження функцій переходів додатковими операціями, обробку взаємодії винесено у внутрішні методи вузла. Кожен об'єкт вузла може мати набір обробників enter, onInput та exit (рис. 3.20), що дозволяє зосередити функції надсилання повідомлень, валідації вхідних даних та оновлення контексту на рівні відповідного стану.

У кожному потоці, що складається з 2 або більше вузлів, повинно бути визначено початковий (initial) та кінцевий вузол (complete). Потік, у якому вузли

зберігаються як колекції, також є об’єктом, який має ідентифікатор, версіонування, та у якому декларативно описуються правила, що регулюють виконання сценарію в цілому (рис. 3.22):

1. TTL (Time-To-Live) – параметр, що визначає час неактивності, після якого виконання сценарію автоматично завершується, а дані потоку видаляються зі сховища сесій.
2. Обробники onEnter, onExit та onError викликаються відповідно під час входу в потік, його завершенні та при виникненні помилки.
3. Глобальні переривання, що обробляються з вищим пріоритетом, ніж onInput у вузлах. Переривання може бути задано як рядок, регулярний вираз, функція та відповідний обробник.

```
return Flow("flow_command_processor")
    .ttl(10 * 60 * 1000)
    .onEnter(async (ctx) => ctx.reply("Command processor started."))
    .onExit(async (ctx) => ctx.reply("Command processor stopped."))

    .interrupt("/pause", async ({ ctx, transition }) => {
        await ctx.reply("Pausing current operation...");
        await transition.pause();
    })
    .interrupt(/^(stop|exit|quit)$/i, async (ctx) => {
        await ctx.reply("Exiting command processor.");
        await transition.abort();
    })

    .node("start", (node) =>
        node
            .initial()
            .enter(async (ctx) => {
                ctx.session.commandCount = 0;
                ctx.session.isAdmin = ctx.sender === admin;
            })
            .next("node_process_command")
    )
```

Рисунок 3.22. Визначення потоку з глобальними правилами та початковим кроком

3.3.4. Перевірка визначень потоків

Побудова надійної бібліотеки передбачає не лише надання інструментів конструювання потоків діалогів, а й механізмів підтвердження їх алгоритмічної коректності до моменту виконання.

Перевірка здійснюється шляхом статичного аналізу структури сценарію, що включає етапи, представлені на рисунку 3.23.

```
export function analyzeFlowGraph(flow) {
    const { adjacency, edges, dynamicResolvers } = createAdjacencyMap(flow);
    const reachable = findReachable(flow, adjacency);
    const unreachableNodes = [];

    for (const nodeId of flow.nodes.keys()) {
        if (!reachable.has(nodeId)) {
            unreachableNodes.push(nodeId);
        }
    }

    const cyclesWithoutExit = evaluateCycles(flow, adjacency);
    const completionPath = hasCompletionPath(flow, adjacency);

    const graph = buildGraphMetadata(flow, edges, dynamicResolvers);

    return {
        unreachableNodes,
        cyclesWithoutExit,
        hasCompletionPath: completionPath,
        graph,
    };
}
```

Рисунок 3.23. Функція, що формує набір характеристик графа потоку

Спочатку виконується побудова матриці суміжності графа. Для кожного вузла збираються всі можливі переходи (default, branches, resolver та timeout), (рис. 3.24) результатом чого є відображення кожного вузла на множину його сусідів, масив ребер графа з інформацією про їх тип та масив вузлів, у яких перехід є динамічним (resolver).

```
'start' => {
  id: 'start',
  enter: [AsyncFunction (anonymous)],
  exit: null,
  onInput: [AsyncFunction (anonymous)],
  timeout: null,
  transitions: {
    default: null,
    branches: [
      { when: [Function: when], to: 'path_a' },
      { when: [Function: when], to: 'path_b' }
    ],
    resolver: null
  },
  complete: false,
  metadata: {},
  initial: true
},
```

Рисунок 3.24. Результати аналізу окремого вузла потоку

На наступному кроці аналізу потоку виконується виявлення недосяжних вузлів через пошук у ширину від початкового вузла потоку. Далі за алгоритмом визначаються цикли, які не мають виходів. Тут для пошуку сильнозв'язаних компонент застосовується алгоритм Тар'яна (рис. 3.25), який виконує рекурсивний обхід графа. У результаті отримується масив SCC, кожна з яких може містити один або більше вузлів.

```
const neighbors = adjacency.get(nodeId) || new Set();
for (const neighbor of neighbors) {
    if (!indexMap.has(neighbor)) {
        strongConnect(neighbor);
        lowlinkMap.set(nodeId, Math.min(lowlinkMap.get(nodeId), lowlinkMap.get(neighbor)));
    } else if (onStack.has(neighbor)) {
        lowlinkMap.set(nodeId, Math.min(lowlinkMap.get(nodeId), indexMap.get(neighbor)));
    }
}

if (lowlinkMap.get(nodeId) === indexMap.get(nodeId)) {
    const component = [];
    let member;
    do {
        member = stack.pop();
        onStack.delete(member);
        component.push(member);
    } while (member !== nodeId && stack.length);

    components.push(component);
}
```

Рисунок 3.25. Фрагмент коду реалізації алгоритму Тар'яна

Цикли є стандартною частиною логіки діалогу, що дозволяє, наприклад, повертати користувача на попередній крок при помилці. Тому кожна виявлена SCC перевіряється за трьома умовами: наявність ребра до вузла поза компонентою, до вузла завершення потоку з прапорцем complete або наявність динамічного обробника, який може змінити напрям переходу в процесі виконання. Помилкою вважається наявність компонент, які не відповідають жодній з цих умов.

На останньому етапі виконується пошук у глибину від початкового вузла до досягенення хоча б одного вузла з прапорцем complete.

Результати роботи цих алгоритмів зберігаються в об'єкті, що включається до структури потоку (рис. 3.26). Зокрема, ці дані придатні для візуалізації та

документування сценаріїв, що в перспективі дає змогу реалізувати графічний редактор.

```
metadata: { description: 'e2e testing flow', tags: [ 'test', 'e2e' ] },
analysis: {
  unreachableNodes: [],
  cyclesWithoutExit: [],
  hasCompletionPath: true,
  graph: {
    initialNode: 'start',
    nodes: [
      ...
    ],
    edges: [
      { from: 'start', to: 'path_a', kind: 'branch' },
      { from: 'start', to: 'path_b', kind: 'branch' },
      { from: 'path_a', to: 'timeout', kind: 'default' },
      { from: 'path_b', to: 'timeout', kind: 'default' },
      { from: 'timeout', to: 'branch_c', kind: 'branch' },
      { from: 'timeout', to: 'branch_d', kind: 'branch' },
      { from: 'timeout', to: 'finish', kind: 'timeout' },
      { from: 'branch_c', to: 'finish', kind: 'default' },
      { from: 'branch_d', to: 'finish', kind: 'default' }
    ],
    completionNodes: [ 'finish' ],
    dynamicResolvers: [],
    interrupts: [
      { index: 0, triggerType: 'string' },
      { index: 1, triggerType: 'string' },
      { index: 2, triggerType: 'function' }
    ]
  }
```

Рисунок 3.26. Фрагмент об’єкта аналізу потоку, на основі якого проводиться валідація

Перевірка структурної цілісності та відповідності опису потоку очікуваному контракту даних реалізована на базі схем бібліотеки Zod. Для демонстрації роботи алгоритму всі переходи, зображені на рисунку 3.27 до вузла «finish» замінено на «start», що створює ряд логічних помилок (рис. 3.27).

```
context: {
  flowId: 'test_flow',
  issues: [
    {
      code: 'custom',
      path: [ 'unreachableNodes' ],
      message: 'Flow graph contains unreachable nodes: finish'
    },
    {
      code: 'custom',
      path: [ 'cyclesWithoutExit' ],
      message: 'Flow graph contains cycles without exit: branch_c -> branch_d -> path_a -> path_b -> start -> timeout'
    },
    {
      code: 'custom',
      path: [ 'hasCompletionPath' ],
      message: 'Flow graph does not contain a path to a completion node'
    }
  ]
},
```

Рисунок 3.27. Помилки, що виникли під час спроби запуску застосунку з некоректно визначеним потоком

3.3.5. Управління та зберігання контексту діалогу

Структура даних сесії оптимізована для мінімізації споживання пам’яті. Для кожного активного діалогового сценарію створюється об'єкт стану, що зберігається у сховищі сесії під заданим користувачем ключем. Варто зазначити, що повна історія листування з користувачем не фіксується, лише технічні параметри переміщення графом сценарію. Об'єкт стану містить поля current (поточний вузол), previous (попередній вузол) та масив history, що зберігає ідентифікатори пройдених вузлів та мітки часу для реалізції логіки таймаутів вузлів (рис. 3.28). Додатково у сесії також зберігаються метадані потоку та довільні користувацькі дані.

Рисунок 3.28. Структура стану потоку, збереженого у сховищі Redis

Забезпечення ізоляції сесій між різними користувачами та акаунтами ботів, яких у застосунку може бути декілька, досягається через генерацію складених ключів, які формуються за шаблоном, що враховує обліковий запис бота, тип чату та ідентифікатор співрозмовника. Для забезпечення конфіденційності усі складові ключів проходять хешування за алгоритмом SHA-256 (рис. 3.29).

```
_getSessionKey(ctx) {
    const accountKey = this._hashIdentifier(ctx.account);
    const userId = ctx.sender?.uuid || ctx.sender?.number;
    const userKey = this._hashIdentifier(userId);

    if (ctx.isGroup) {
        const groupKey = this._hashIdentifier(ctx.group.id);
        return `${accountKey}:group:${groupKey}:${userKey}`;
    }

    return `${accountKey}:private:${userKey}`;
}

_hashIdentifier(identifier) {
    const rawId = identifier === undefined || identifier === null
        ? "unknown" : String(identifier);
    const digest = crypto.createHash("sha256")
        .update(rawId).digest("hex");
    return `h-${digest.slice(0, 16)}`;
}
```

Рисунок 3.29. Формування хешованих ключів сесій з розділенням за типом діалогу

Крім того, за умови надання розробником секретного ключа, дані сесій шифруються за алгоритмом AES-256-GCM, який забезпечує як конфіденційність, так і цілісність даних через тег автентифікації. Ключ шифрування формується з користувацького секрету за допомогою функції деривації ключа PBKDF2 з використанням HMAC-SHA256 та фіксованої солі. Вектор ініціалізації генерується випадковим чином для кожного запису окремо (рис. 3.30).

```
encrypt(serialized, method = "_encrypt") {
    try {
        const iv = crypto.randomBytes(ENCRYPTION_DEFAULTS.IV_LENGTH);
        const cipher = crypto.createCipheriv(ENCRYPTION_DEFAULTS.ALGORITHM, key, iv);

        const encrypted = Buffer.concat([
            cipher.update(serialized, "utf8"),
            cipher.final(),
        ]);

        const authTag = cipher.getAuthTag();

        return {
            iv: iv.toString("base64"),
            data: encrypted.toString("base64"),
            authTag: authTag.toString("base64"),
        };
```

Рисунок 3.30. Шифрування даних сесій з використанням AES-256-GCM

Оскільки час життя сесії та окремого потоку може бути різним, у розширенні Flow було реалізовано метод для синхронізації TTL для того, аби сесія не була видалена раніше за найдовший активний потік (рис. 3.31).

Рисунок 3.31. Час життя звичайної сесії та такої, що містить активний потік

Для цілей тестування та попереднього налагодження персональних асистентів у бібліотеці передбачено два вбудованих типи сховищ: у пам’яті застосунку як колекції та у вигляді окремих файлів з розширенням json у локальній файловій системі. Для готових до експлуатації застосунків пропонується адаптер для бібліотеки Keyv, що дозволяє використовувати різні зовнішні сховища даних (Redis, MongoDB, PostgreSQL, SQLite). Альтернативно, розробник може імплементувати власну стратегію зберігання та передати її до менеджера сесій.

3.4. Тестування та демонстрація роботи

У рамках даної роботи, як і передбачалось, було проведено модульне, інтеграційне та наскрізне тестування основних класів, алгоритмів та функцій бібліотеки. Рівень покриття тестами не є високим, але достатнім для підтвердження працездатності основного функціоналу бібліотеки (рис. 3.32).

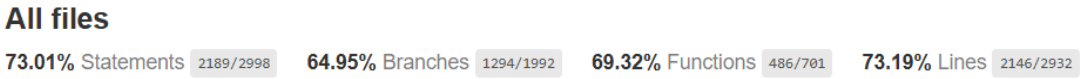


Рисунок 3.32. Звіт про відсоток покриття тестами, згенерований Jest

Модульними тестами було перевірено той функціонал, який за очікуваннями буде найчастіше використовуватись персональними асистентами. Зокрема, у ядрі бібліотеки перевірено компонування об’єктів для Signal-cli, нормалізацію вхідних даних, маскування логів тощо (рис. 3.33).

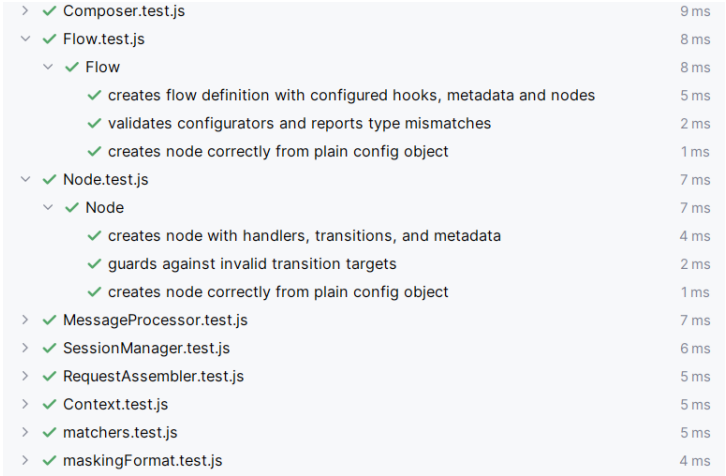


Рисунок 3.33. Фрагменти результатів виконання модульних тестів

Також, було перевірено компонування ланцюгів обробників (Composer), створення контексту повідомлення та визначення адресата відповідей для різних типів діалогів (Context), розпізнавання команд та співставлення тексту з регулярними виразами для маршрутизації (matchers) та деякі деталі роботи інших розширень.

Основні ж зусилля було зосереджено навколо плагіну Flow, який формує відмінну рису бібліотеки порівняно з наявними рішеннями: побудова вузлів та сценаріїв, валідація об’єкту конфігурації потоку (створення матриці суміжності, знаходження недосяжних вузлів та замкнених циклів), керування життєвим циклом потоку (рис. 3.34).

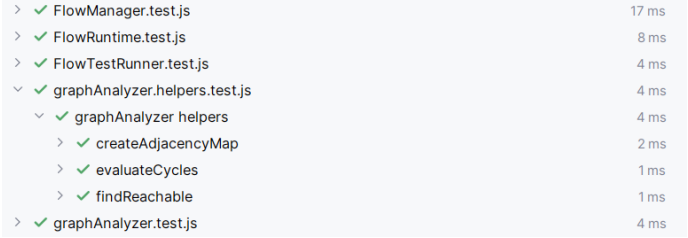


Рисунок 3.34. Тестування плагіну Flow

Окрім цього, перевірено роботу машини станів та підтверджено, що раніше описана логіка з декількома типами переходів та їх пріоритетність працює згідно з очікуваннями (рис. 3.35).

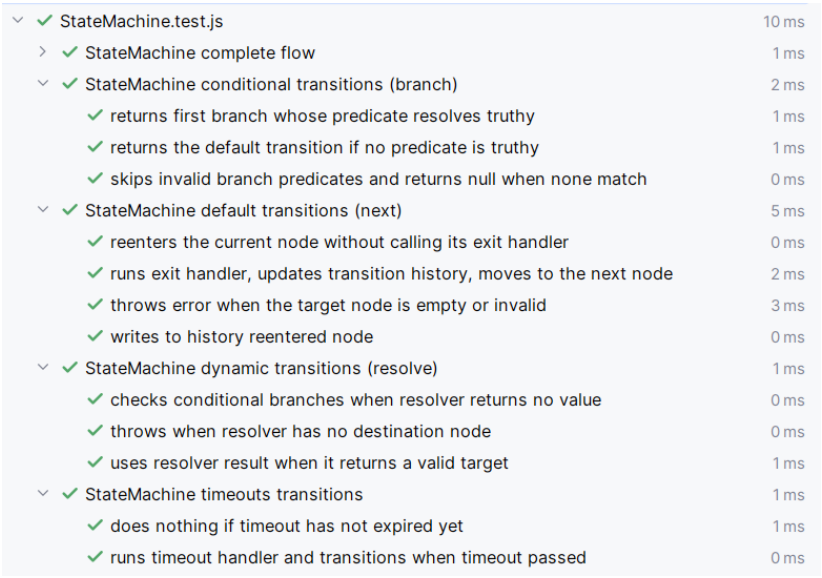


Рисунок 3.35. Тестування роботи машини станів

Для виконання інтеграційних тестів було розроблено набір додаткових утиліт, які також експортуються як частина публічного API бібліотеки для використання розробниками у власних застосунках.

Таблиця 3.2

Допоміжні утиліти для тестування

Утиліта	Призначення та функції
createMockCtx	Фабрика для імітації контексту повідомлення, включно з інформацією про відправника, групу, мок-реалізаціями методів reply та допоміжних перевірок hasText, isPrivate тощо
createMockExecutor	Фабрика для підміни виконавців для імітації з'єднання з Signal-cli. Дозволяє налаштувати черги відповідей та помилок

Таблиця 3.2 (продовження)

Допоміжні утиліти для тестування

Утиліта	Призначення та функції
FlowTestRunner	Тестовий виконавець діалогових сценаріїв, який налаштовує ланцюг обробників, сесій, надає методи для керування життєвим циклом потоку і симуляції проходження повідомлень через обробники
createMockExecCtx	Поєднує мок-контекст з підміненими реалізаціями переходів між вузлами потоку для ізольованого тестування обробників окремих вузлів

Інтеграційними тестами було перевірено процес проходження повідомлення через послідовність проміжних обробників з маршрутизацією до відповідного кінцевого обробника, виконання багатокрокового сценарію з різними типами переходів між вузлами та збереженням проміжного стану в сесії, обробку функцій переривань потоку, синхронізацію TTL тощо (рис. 3.36).

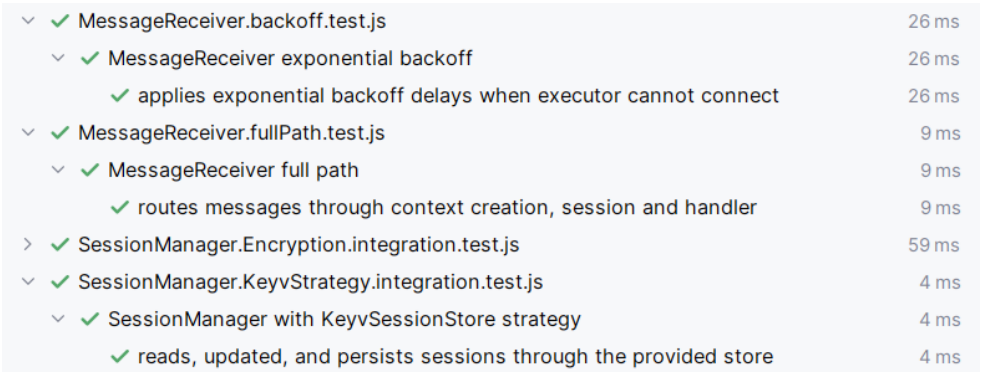


Рисунок 3.36. Тестування функціональності плагіну Messaging та сесій

Опис наскрізного тестування одночасно слугуватиме демонстраційною частиною роботи. Для його проведення було використано два реальні облікові записи Signal: тестованого бота (далі – SUT, System under test) та додаткового акаунту (тестувальник), який також приєднано до Signal-cli.

Обидва облікові записи використовуватимуть API бібліотеки з різними режимами підлючення до Signal-cli, проте SUT використовуватиме повний набір плагінів (Messaging, Session, Middleware, Context) з підключенням до Redis, а скрипт тестувальника – лише ядро бібліотеки та Jest. Тестування виконувалось на потоці, конфігурація якого зображена на рисунку 3.26.

Скрипт тестувальника перевіряв роботу застосунку як «чорну скриньку», взаємодіючи з SUT виключно через інтерфейс месенджера, що дозволяє перевірити роботу бібліотеки в умовах, наближених до реальної експлуатації.

У першій групі тест-кейсів перевірено роботу проміжних обробників та компонента Router. Відповіді бота на команди «/start» та «/status» підтвердили коректність роботи ланцюга обов’язків, сформованого класом Composer. Також підтверджено коректну роботу класу Context при створенні об'єкта контексту та методів відправки відповіді і визначення адресата (рис. 3.37).

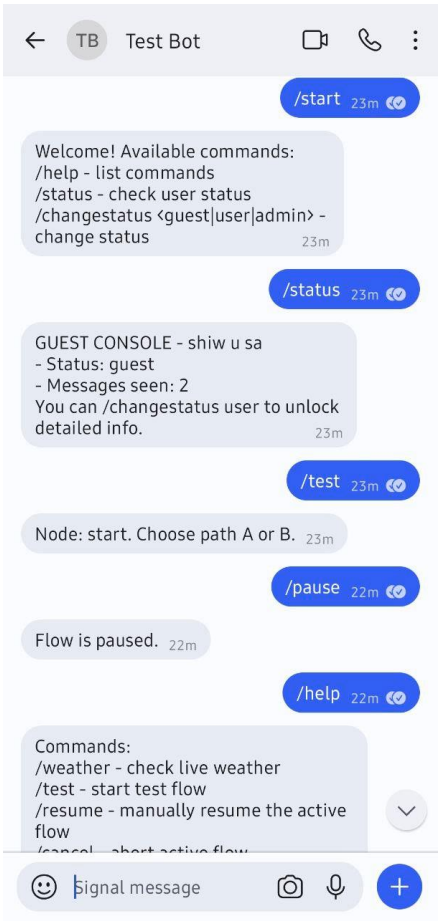


Рисунок 3.37. Демонстрація роботи маршрутизаторів повідомлень, а також функцій переривань потоків

Командами «/test» та «/pause» було перевірено запуск та призупинення сценарію діалогу з боку користувача, що підтвердило пріоритетність глобальних переривань над локальною логікою вузлів, а командою «/help», яка не належить до логіки потоку, коректну роботу маршрутизації повідомлень, коли виконання сценарію поставлено на паузу.

Зокрема, було протестовано механізми умовної маршрутизації (методи when, and, or), що змінюють доступність команд залежно від стану сесії користувача: з порівняння рисунків 3.37 та 3.38 видно, що на останньому за команди «/status» змінилась роль користувача з guest до admin, оскільки була створена сесія для потоку.

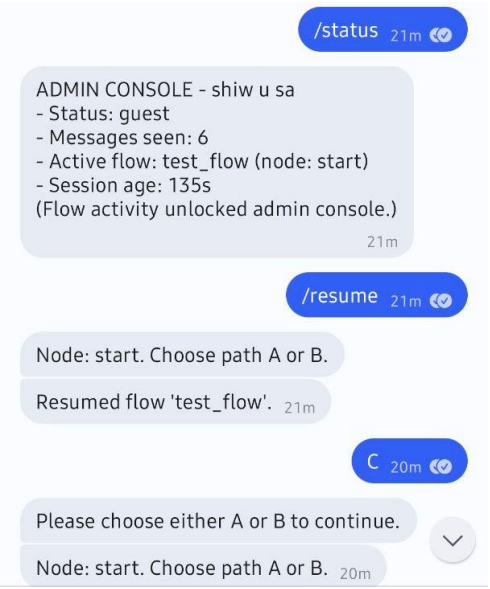


Рисунок 3.38. Повторна перевірка статусу, продовження виконання сценарію

Командою «/resume» перевірено роботу машини станів: відновлення призупиненого потоку на тому ж кроці, повторний вхід у поточний вузол при введені невалідних даних, а також стан сесії (рис. 3.39).

```
"state": {
  "active": true,
  "paused": false,
  "pauseReason": null,
  "current": "start",
  "previous": "start",
  "history": [
    "start",
    "start"
  ],
  "nodeEnteredAt": 1764775005225,
  "startedAt": 1764774929908,
  "pausedAt": 1764774939954,
  "resumedAt": 1764774976827
},
"data": {}
```


Рисунок 3.39. Повторний запис входу у стан діалогу

В наступних тестах скрипт тестувальника продовжує роботу в тому ж сценарії, очікує на спливання таймауту і перевіряє, що потік було завершено з очікуваною історією кроків (рис. 3.40).

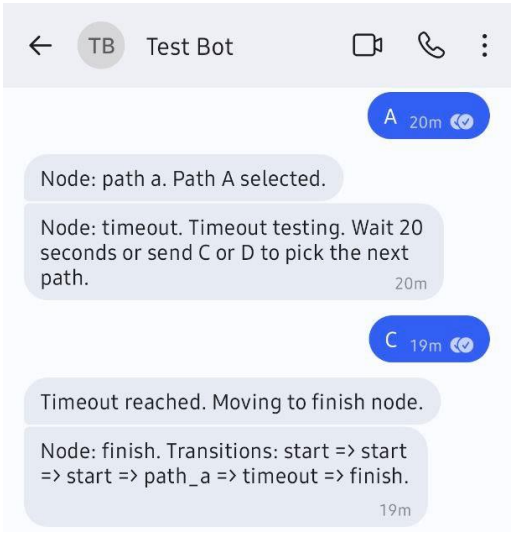


Рисунок 3.40. Завершення виконання потоку через спливання часу очікування

Остання група тестів фокусувалась на перевірці надійності логіки сесій. Надсилання команд у груповий чат одразу після зміни статусу в приватному діалозі перевірило коректність функції генерації ключів сесій, що підтверджує ізоляцію даних не лише між різними користувачами, а й різними діалогами (рис. 3.41).

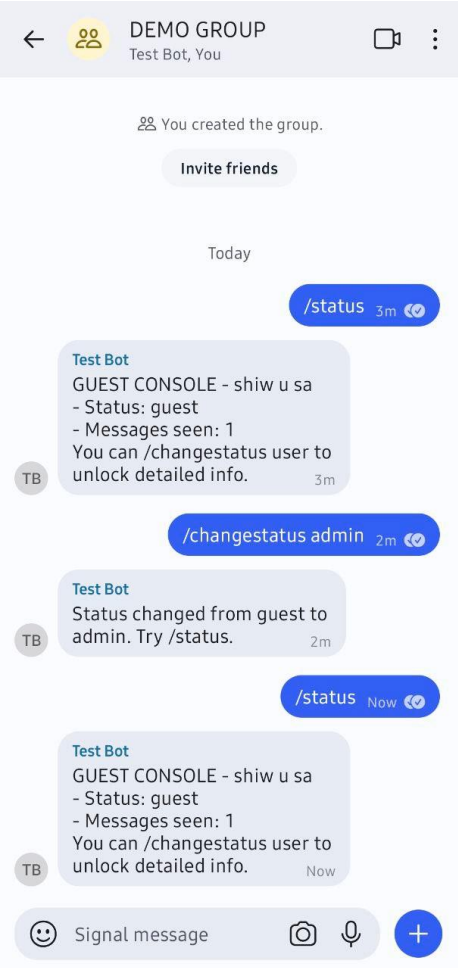


Рисунок 3.41. Комунікація з персональним асистентом у груповому чаті

Останньою командою «/status» перевірено інтеграцію з зовнішнім сховищем даниз через адаптер Keyv. Після встановленого періоду неактивності драйвер сховища успішно видалив запис, а менеджер сесій бібліотеки коректно обробив цю ситуацію, ініціювавши нову сесію.

Результати проходження всіх описаних кроків зображено на рисунку 3.42.

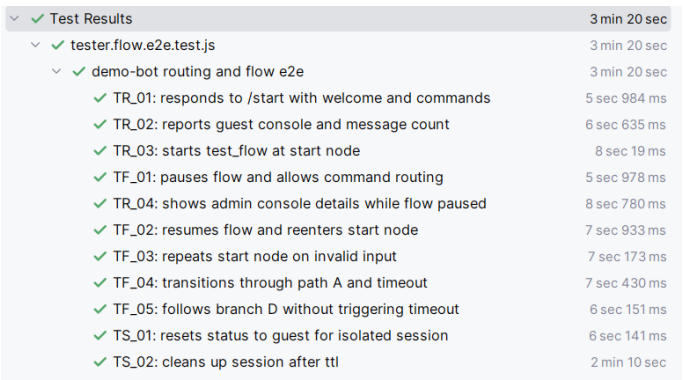


Рисунок 3.42. Тест-кейси наскрізного тестування

Тож, ці тести демонструють працездатність бібліотеки і можливість створення персональних асистентів для Signal навіть за умови відсутності офіційного API.

ВИСНОВОК ДО ТРЕТЬОГО РОЗДІЛУ

Третій розділ було присвячно опису основних технічних деталей та загального принципу роботи розробленої бібліотеки для створення персональних асистентів у месенджері Signal.

Бібліотека бере на себе відповідальність за роботу з інтерфейсом командного рядка Signal-cli, пропонуючи зручний програмний інтерфейс та при цьому потребуючи мінімальної конфігурації для старту. Крім того, вона надає декілька додаткових розширень (а також дозволяє підключати власні), які мають на меті спростити процес побудови діалогових помічників і скоротити час на розроблення. У результаті виконаної у межах цього проєкту роботи розробник отримує:

- Знайомий API, який використовується у більшості подібних бібліотек;
- Вибір серед різних варіантів способу запуску Signal-cli, стратегії отримання повідомлень та сховища сесійних даних;
- Зручні методи для роботи з повідомленнями, фільтрацію непотрібних вхідних даних;
- Вичерпний набір функціоналу для налаштування та подальшої роботи персонального асистента;
- Можливість створювати складні сценарії діалогу з різними методами визначення наступного кроку;
- Методи формування відповіді персонального асистента на основі правил, включаючи зіставлення вхідного тексту за ключовими словами, регулярними виразами чи за типом події;
- Утиліти для тестування діалогового помічника без реального підключення до серверів Signal.

Використані у бібліотеці методи експоненційної затримки, адаптивного опитування та перевірка визначень струтур графів забезпечують стабільну роботу, а проведене на останньому етапі тестування підтверджує, що для

закритого до автоматизації Signal все ж таки можна створювати повноцінних діалогових помічників, які за своїми функціональними можливостями не поступаються рішенням на платформах із відкритим API.

РОЗДІЛ 4
РОЗРОБЛЕННЯ СТАРТАП-ПРОЄКТУ

Створення програмного продукту вимагає не лише інженерних рішень, але й оцінки його комерційної життєздатності. Цей розділ презентує розробку стартап-проєкту на основі створеної бібліотеки. Проводиться комплексний аналіз ринкових можливостей, конкурентного середовища та потенційних загроз для обґрунтування стратегії комерціалізації.

4.1. Опис ідеї проєкту

У таблицях 4.1 та 4.2 представлено, відповідно, опис ідеї стартап-проєкту із визначенням напрямків застосування та вигод для користувача, а також порівняльний аналіз сильних, слабких та нейтральних характеристик розробки відносно існуючих конкурентних рішень.

Таблиця 4.1

Опис ідеї стартап-проєкту

Зміст ідеї	Напрямки застосування	Вигоди для користувача
Створення бібліотеки, що надає розробникам інструменти для побудови персональних асистентів у месенджері Signal	Внутрішньоорганізаційні комунікаційні системи	Автоматизація внутрішніх процесів обміну даними в єдиному, повністю захищеному корпоративному каналі
	Клієнтоорієнтовані корпоративні сервіси	Побудова довірчих відносин з клієнтами через гарантовану конфіденційність комунікацій, захищених від втручання третьої сторони або платформи
	Персональні автоматизаційні рішення	Зменшення часу на виконання повторюваних завдань та забезпечення контролю над особистими даними

Таблиця 4.2

Визначення сильних, слабких та нейтральних характеристик ідеї проєкту

№ п/п	Характеристика	Власна розробка	Конкуренти				W	N	S
			1	2	3	4			
1	Кількість підтримуваних операцій Signal	+	-	-	+	+		+	
2	Ергономічність API	+	+	-	+	+		+	
3	Робота з різними типами повідомлень	+	+	-	+	+		+	
4	Синтаксичні розширення для роботи з повідомленнями	+	+	+	-	+		+	
5	Інструменти для управління потоком діалогу	+	-	-	-	-			+
6	Зручність початкового налаштування	+	-	+	+	-		+	
7	Популярність марки	-	+	-	+	-	+		
8	Підтримка мультиакаунтності	+	-	-	+	+			+
10	Вичерпна документація API	+	-	-	+	+	+		

Таблиця 4.2 (продовження)

Визначення сильних, слабких та нейтральних характеристик ідеї проєкту

№ п/п	Характеристика	Власна розробка	Конкуренти				W	N	S
			1	2	3	4			
11	Тестованість користувацького коду	+	+	-	-	-			+

4.2. Технологічний аудит ідеї проєкту

Технологічний аудит, представлений у таблиці 4.3, аналізує стек технологій, необхідний для реалізації ідеї проєкту. Цей перелік охоплює як технології, що безпосередньо використані для створення поточної версії бібліотеки, так і ті, що розглядаються для впровадження на етапах подальшого розвитку продукту.

Таблиця 4.3

Технологічна здійсненність ідеї проекту

№ п/п	Ідея проєкту	Технології її реалізації	Наявність технологій	Доступність технологій
1	Програмна взаємодія з месенджером Signal	Signal-cli, Docker	Наявні	У відкритому доступі
2	Збереження сесійних даних	Keyv	Наявні	У відкритому доступі

Таблиця 4.3 (продовження)

Технологічна здійсненність ідеї проекту

№ п/п	Ідея проєкту	Технології її реалізації	Наявність технологій	Доступність технологій
3	Логування	Winston	Наявні	У відкритому доступі
4	Сервіс опитування сервера Signal	Адаптивне опитування з експоненційною затримкою	Власна розробка	Доступні безкоштовно
5	Інструментарій для тестування асистентів	Jest	Наявні	У відкритому доступі
6	Стратегія управління діалогом	Модель скінченних автоматів	Власна розробка	Доступні безкоштовно
7	Власний Signal клієнт	libsignal-client, libsignal-service-java	Наявні	У відкритому доступі
8	Візуальний конструктор потоків діалогів	React Flow	Наявні	У відкритому доступі
9	NLU для вибору відповіді	DialogFlow/Rasa/Wit	Наявні	DialogFlow/Rasa платні, Wit безкоштовний
10	Управління чергою повідомлень	BullMQ, p-queue.js	Наявні	У відкритому доступі

Технологічна реалізація проєкту є цілком можливою, оскільки базовий набір технологій (№1-6) доступні та засвоєні. Для ідеї №9 вибір сервісу не є

однозначним: платформи на кшталт DialogFlow та Rasa у безкоштовних тарифах встановлюють обмеження, тоді як повністю безоплатні інструменти (наприклад, Wit) використовують дані для тренування глобальних моделей.

4.3. Аналіз ринкових можливостей запуску стартап-проєкту

Наступним кроком є аналіз ринкових можливостей та загроз, що визначать комерційний успіх продукту. Цей етап дозволяє спланувати стратегію виходу на ринок, враховуючи стан конкурентного середовища, специфічні потреби потенційних клієнтів та бар'єри входу. У таблиці 4.4 наведено попередню характеристику потенційного ринку для розробленої бібліотеки.

Таблиця 4.4

Попередня характеристика потенційного ринку стартап-проєкту

№ п/п	Показники стану ринку (найменування)	Характеристика
1	Кількість головних гравців	4
2	Загальний обсяг продаж, грн/ум.од.	-
3	Динаміка ринку (якісна оцінка)	Зростає
4	Наявність обмежень для входу (характер обмежень)	Високі. Характер: технічні (відсутність API Signal), криптографічні (складність протоколу), вартість підтримки (часті оновлення Signal)
5	Специфічні вимоги до стандартизації та сертифікації	Немає
6	Середня норма рентабельності в галузі або по ринку, %	15%

Середня норма рентабельності у загальному рикнку оцінюється у 15%. Цей показник перевищує середні банківські відсотки на вкладення в Україні (приблизно 9.7%). Оскільки потенційна дохідність проєкту вища за альтернативні, менш ризиковані інструменти інвестування, можна зробити попередній висновок, що ринок є економічно привабливим для входження.

Для поглиблення ринкового аналізу, в таблиці 4.5 визначено ключові цільові сегменти. Розуміння аудиторії є неповним без оцінки зовнішнього середовища, тому в таблиці 4.6 та таблиці 4.7 відповідно ідентифіковано головні фактори загроз та можливостей, які безпосередньо впливають на стратегію впровадження проєкту.

Таблиця 4.5

Характеристика потенційних клієнтів стартап-проєкту

№ п/п	Потреба, що формує ринок	Цільова аудиторія	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
1	Автоматизація внутрішніх корпоративних процесів	Середні та великі підприємства у галузях з високими вимогами до безпеки, DevOps команди	Потреба мінімізувати ризики безпеки при впровадженні автоматизації; пошук рішень, що інтегруються з існуючою інфраструктурою	Надійність, простота інтеграції з корпоративними системами, стабільність роботи, комерційна підтримка

Таблиця 4.5 (продовження)

Характеристика потенційних клієнтів стартап-проєкту

№ п/п	Потреба, що формує ринок	Цільова аудиторія	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
2	Створення клієнтських сервісів (підтримка, консультації) з гарантованою приватністю комунікації з боку платформи	Компанії та організації, що надають конфіденційні послуги	Дотримання регуляторних вимог; прагнення підвищити довіру клієнтів за рахунок використання захищеного каналу	Простота розробки ботів, можливість інтеграції з внутрішніми сервісами, гарантії безпеки обробки даних
3	Автоматизація особистих рутинних завдань	Технічно орієнтовані індивідуальні користувачі	Прагнення максимальної приватності та контролю	Легкість використання, ціна, документація

Таблиця 4.6

Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
1	Залежність від стабільності та	Будь-які суттєві зміни в політиці безпеки з боку	Постійний моніторинг оновлень клієнта та

Таблиця 4.6 (продовження)

Фактори загроз

№ п/п	Фактор	Зміст загрози	Можлива реакція компанії
	політики платформи Signal	Signal Foundation можуть призвести до повної або часткової непрацездатності бібліотеки	протоколу Signal. Формування команди для оперативної адаптації бібліотеки до змін у платформі
2	Низький рівень попиту на комерційний продукт у ніші	Компанії та розробники можуть виявитися не готовими платити за комерційну ліцензію; кількість користувачів Signal менша, ніж у інших месенджерах	Акцент у маркетингових матеріалах на унікальних перевагах. Надання якісної технічної підтримки як частини комерційної пропозиції
3	Поява офіційного API від Signal	Користувачі можуть перейти на офіційне, більш стабільне рішення	Швидка адаптація для роботи поверх офіційного API. Акцент на унікальних можливостях бібліотеки (управління станом, сценарії, візуальний конструктор)

Таблиця 4.7

Фактори можливостей

№ п/п	Фактор	Зміст можливості	Можлива реакція компанії
1	Відсутність прямих конкурентів з аналогічним функціоналом	Існуючі аналоги мають обмежений функціонал	Позиціонування як преміум-рішення для професійної розробки. Акцент у маркетингу на унікальних функціях
2	Зростання глобального попиту на приватність та безпеку комунікацій	Все більше компаній та індивідуальних користувачів шукають альтернативи месенджерам, що збирають великі обсяги даних. Signal є лідером у сегменті безпечних комунікацій	Маркетингове просування рішення як інструменту для створення асистентів із пріоритетом на конфіденційності
3	Високий потенціал для створення екосистеми навколо бібліотеки	Подальший розвиток продукту шляхом створення додаткових розширень та інструментів	Розробка чіткої дорожньої карти розвитку продукту. Залучення спільноти розробників. Поступове розширення функціоналу відповідно до потреб ринку та відгуків клієнтів

У таблицях 4.8, 4.9 та 4.10 представлено комплексний аналіз конкуренції: від визначення типу ринку та аналізу галузевих сил за моделлю М. Портера до обґрунтування конкретних факторів, що забезпечуватимуть

конкурентоспроможність розробленої бібліотеки. Конкуретний аналіз завершується таблицею 4.11, яка переводить раніше визначені якісні фактори у кількісну площину.

Таблиця 4.8

Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
Тип конкуренції: монополістична	На ринку існують продукти, які є схожими, але не ідентичними	Диференціація продукту через унікальний функціонал
За рівнем конкурентної боротьби: глобальний	Інструменти для розробників не має географічних кордонів	Англомовна документація, підтримка та маркетингові матеріали. Моніторинг міжнародних конкурентних рішень
За галузевою ознакою: внутрішньогалузева	Конкуренція відбувається серед інструментів для автоматизації саме в месенджері Signal.	Необхідно враховувати специфічні особливості галузі
Конкуренція за видами товарів: товарно-видова	Конкуренція відбувається між різними реалізаціями одного й того ж типу продукту	Постійне вдосконалення продукту, додавання нових функцій. Акцент на якості реалізації, надійності та документації

Таблиця 4.8 (продовження)

Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
За характером конкурентних переваг: нецінова	Унікальний функціонал, якість, зручність використання та комерційна підтримка	Додання нових функцій до системи, а також вдосконалення вже існуючих, своєчасне виправлення наявних недоліків
За інтенсивністю: не марочна	Користувачі обирають інструменти переважно на основі технічних характеристик, а не лояльності до певного бренду	Активне просування технічних переваг продукту

Таблиця 4.9

Аналіз конкуренції в галузі за М. Портером

	1. Прямі конкуренти в галузі	2. Потенційні конкуренти	3. Постачальники	4. Клієнти	5. Товари замінники
Складові аналізу	Signalbot, Signal-bot, Signal-cli-rest-api,	Signal Foundation, нові стартапи і	Signal Foundation, розробники Signal-cli (для MVP)	Великий, середній та малий бізнес,	Простіші рішення, розробка власного клієнта

Таблиця 4.9 (продовження)

Аналіз конкуренції в галузі за М. Портером

Складові аналізу	1	2	3	4	5
	Signal-rest-ts	комерційні рішення		приватні користувачі	
Висновки	Низький/Середній тиск. Конкуренти переважно безкоштовні, але не задовольняють потребу	Низька загроза	Проєкт повністю залежить від стабільності платформи Signal та її політики щодо сторонніх рішень	Клієнти мають вибір між платформами. Цінова чутливість існує	Замінники (інструменти для інших платформ) є доступними та часто простішими у використанні

Галузь розробки інструментів для Signal є привабливою для входження через слабкість прямих конкурентів у ніші складних асистентів та низьку ймовірність появи офіційного API. Однак вона характеризується дуже високою залежністю від самої платформи Signal та помірною загрозою з боку товарів-замінників (інструментів для менш безпечних, але більш популярних платформ). Фактором успіху є фокус на унікальних перевагах та здатності швидко адаптуватися до змін у Signal.

Таблиця 4.10

Обґрунтування факторів конкурентоспроможності

№ п/п	Фактор конкурентоспроможності	Обґрунтування
1	Унікальний функціонал для складних асистентів	Надання стратегії для керування діалогом та

Таблиця 4.10 (продовження)

Обґрунтування факторів конкурентоспроможності

№ п/п	Фактор конкурентоспроможності	Обґрунтування
		методів формування відповіді
2	Зручний API	Пропонує інтуїтивно зрозумілий API, схожий до звичних популярних бібліотек, що суттєво спрощує та прискорює процес розробки
3	Комерційна підтримка та надійність	Професійна технічна підтримка та гарантії стабільності
4	Фокус на безпеці та приватності платформи Signal	Відповідність глобальному тренду на посилення вимог до конфіденційності даних
5	Потенціал розвитку екосистеми	Можливість побудови цілісної екосистеми навколо продукту

Таблиця 4.11

Порівняльний аналіз сильних та слабких сторін стартап-проєкту

№ п/п	Фактор конкурентоспроможності	Бали 1-20	Порівняння рейтингу товарів конкурентів						
			-3	-2	-1	0	+1	+2	+3
1	Унікальний функціонал	18		+					

Таблиця 4.11 (продовження)

Порівняльний аналіз сильних та слабких сторін стартап-проєкту

№ п/п	Фактор конкурентоспроможності	Бали 1-20	Порівняння рейтингу товарів конкурентів						
			-3	-2	-1	0	+1	+2	+3
2	Фокус на безпеці Signal	17				+			
3	Зручний API	15			+				
4	Комерційна підтримка та надійність	13	+						
5	Потенціал розвитку	11			+				

У таблиці 4.12 систематизовано внутрішні та зовнішні фактори проєкту. Ці результати слугують основою для визначення та порівняння практичних альтернатив впровадження проєкту, які детально розглянуто у таблиці 4.13.

Таблиця 4.12

SWOT-аналіз стартап-проєкту

Сильні сторони:	Слабкі сторони:
Унікальність функціоналу Зручний API, легке початкове налаштування Комерційна підтримка та надійність	Новий, невідомий бренд Є потреба у часі для виходу на ринок Комерційна ціна проти безкоштовних аналогів
Можливості:	**Загрози:**
Відсутність конкурентів з аналогічним функціоналом Зростання попиту на приватність Потреба ринку у складній автоматизації Потенціал розвитку екосистеми	Зміни у протоколі/політиці Signal Поява офіційного API від Signal Низький попит на платне рішення у ніші Швидший або успішніший вихід на ринок нового конкуруючого рішення

Таблиця 4.13

Альтернативи ринкового впровадження стартап-проєкту

№ п/п	Альтернатива ринкової поведінки	Ймовірність отримання ресурсів	Терміни реалізації
1	Розповсюдження як платного програмного продукту з різними рівнями ліцензування	Середня (50-70%). Потребує або початкових інвестицій для фінансування команди розробки та маркетингу, або швидкого залучення перших платоспроможних клієнтів	6-12 місяців
2	Випуск базової версії (ядра) під відкритою ліцензією. Розробка та продаж розширених комерційних модулів	Низька (20-30%) для успішної монетизації преміум-модулів. Модель дозволяє швидше побудувати користувацьку базу та впізнаваність бренду	3-6 місяців для відкритої версії. 12+ місяців для розробки та виведення на ринок розширень
3	Використання бібліотеки для надання послуг з розробки кастомних асистентів Signal для корпоративних клієнтів	Висока (90%) за умови активного пошуку клієнтів. Ця модель дозволяє отримувати дохід майже одразу, використовуючи MVP у наданні послуг	3-6 місяців

Згідно з критеріями (вища ймовірність отримання ресурсів та коротші строки реалізації), остання альтернатива виглядає найбільш привабливою для початкового етапу стартапу. Вона дозволяє швидко перевірити ринковий попит, отримати перший дохід та відгуки, використовуючи вже наявний MVP, проте потребує залучення додаткових людських ресурсів. Отриманий досвід та кошти

могли б бути надалі реінвестовані у розробку продукту за моделлю комерційної ліцензії.

4.4. Розроблення ринкової стратегії продукту

У таблицях 4.14 – 4.17 представлено розробку маркетингової програми стартап-проєкту, що охоплює три ключові складові комплексу маркетингу. Спочатку визначено характеристики продукту, проаналізовано його основні переваги (табл. 4.14) та побудовано трирівневу модель (табл. 4.15). Далі подано обґрунтування цінової політики (табл. 4.16) і розроблено систему збуту (табл. 4.17), спрямовану на ефективне донесення продукту до цільових сегментів ринку.

Таблиця 4.14

Вибір цільових груп потенційних споживачів

№ п/п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивні сть конкуренці ї в сегменті	Простота входу у сегмент
1	Компанії та організації у галузях з високими вимогами до захисту даних	Висока	Середній. Попит зростає через посилення законодавства щодо захисту даних та політик безпеки у межах компаній	Низька. Прямі конкуренти Не мають достатнього функціоналу, а інші платформи не є настільки захищеними	Середня. Потребує доведення надійності та безпеки

Таблиця 4.14 (продовження)

Вибір цільових груп потенційних споживачів

№ п/п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивні сть конкуренці ї в сегменті	Простота входу у сегмент
2	Технічні команди розробників	Висока	Середній. Кількість розробників та користувачів у Signal менша	Низька	Легка. Потребує якісної технічної документації, прикладів та активної взаємодії зі спільнотою
3	Технічно орієнтовані індивідуальні користувачі	Середня	Середній.	Висока. Безкоштовні аналоги є більш привабливими, навіть з меншим функціоналом	Легка. Потребує встановлення низької ціни та якісної технічної документації

Обрані цільові групи: компанії у галузях з високими вимогами до захисту даних, технічні команди, індивідуальні користувачі (з урахуванням специфіки ціноутворення для цього сегменту).

Таблиця 4.15

Визначення базової стратегрії розвитку

№ п/п	Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможі позиції відповідно до обраної альтернативи	Базова стратегія розвитку
1	Комерційна ліцензія на використання бібліотеки	Диференційований маркетинг	Унікальний функціонал, фокус на безпеці платформи Signal	Стратегія спеціалізації

Таблиця 4.16

Визначення базової стратегії конкурентної поведінки

№ п/п	Чи є проект «першопрохідцем» на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурен та, і які?	Стратегія конкурентної поведінки
1	Ні	Комбінований підхід	Ні. Основна цінність полягає в унікальних характеристиках	Стратегія заняття конкурентної ніші

Таблиця 4.17

Визначення стратегії позиціонування

№ п/п	Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкуренто спроможні позиції власного стартап проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту
1	Надійність, безпека, простота інтеграції, комерційна підтримка, інтуїтивно зрозумілий API, документація	Стратегія спеціалізації	Унікальний функціонал, фокус на безпеці Signal, API, що приховує складність взаємодії,	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту

4.5. Розроблення маркетингової програми стартап-проєкту

Опираючись на розроблену ринкову стратегію та позиціонування, у цьому підрозділі детально викладається маркетингова програма проєкту. У таблицях 4.18 та 4.19 проаналізовано характеристики та переваги продукту, далі обґрунтовано політику ціноутворення у таблиці 4.20 та сформовано систему збуту у таблиці 4.21. Проведення маркетингового аналізу завершено описом концепції маркетингових комунікацій у таблиці 4.22.

Таблиця 4.18

Визначення ключових переваг концепції потенційного товару

№ п/п	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити
1	Створення складних діалогових асистентів у Signal	Надання зручних інструментів для реалізації складної логіки ботів	Відсутність таких функцій у конкурентів
2	Забезпечення максимальної приватності та безпеки комунікацій при автоматизації	Можливість створювати асистентів на базі Signal, що гарантує E2E-шифрування та відсутність збору метаданих платформою	Використання переваг Signal: на відміну від інструментів для Telegram/WhatsApp, проєкт дозволяє будувати рішення з найвищим рівнем конфіденційності, що є унікальною перевагою для окремих галузей
3	Спрощення та прискорення розробки для Signal, враховуючи відсутність офіційного API	Зручний програмний інтерфейс, що бере на себе технічні деталі, та плагінна архітектура для гнучкості	Кращий досвід розробника. Підтримка як додаткова перевага над доступними аналогами

Таблиця 4.19

Опис трьох рівнів моделі товару

Рівні товару	Сутність та складові		
I. Товар за задумом	Бібліотека для створення персональних асистентів у месенджері Signal		
II. Товар у реальному виконанні	Властивості/характеристики	М/Нм	Вр/Тх/Тл/Е/Ор
	Різні методи формування відповіді	М	Тх
	Стратегія керування діалогом	М	Тх
	Зручний програмний інтерфейс	Нм	Е
	Можливість підключати розширення	Нм	Тх
	Інструментарій для тестування ботів	М	Тх
	Візуальний конструктор	Нм	Е
	Якість: тестованість бібліотеки і коду користувача; документація; підтримка		
	Пакування: пакет npm для Node.js		
	Марка: SiBLi		
III. Товар із підкріпленням	До продажу: приклади використання, документація		
	Після продажу: комерційна технічна підтримка; регулярні оновлення		
За рахунок чого потенційний товар буде захищено від копіювання: авторське право, розповсюдження на умовах ліцензії; спеціалізовані алгоритми взаємодії з месенджером Signal, інструментарій для тестування прикладного коду, поєднання різних розширень, що ускладнює швидке відтворення всієї функціональності продукту.			

Таблиця 4.20

Визначення меж встановлення ціни

№ п/п	Рівень цін на товари замінники	Рівень цін на товари аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі встановлення ціни на то вар/послугу
1	Базовий рівень часто безкоштовний	0$	Компанії: Середній/Високий. Технічні команди: Середній. Індивідуальні розробники: Низький/Середній.	Нижня межа: 49$/рік для індивідуальних розробників Верхня межа: 499$/рік для корпоративних ліцензій

Таблиця 4.21

Формування системи збуту

№ п/п	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
1	Компанії у галузях з високими вимогами до конфіденційності**:** ретельна технічна оцінка перед придбанням, вимога тестування та аудиту	Надання демонстрацій продукту та документації, консультації щодо інтеграції	Канал нульового рівня	Прямі продажі через корпоративний сайт, персональні комерційні пропозиції,

Таблиця 4.21 (продовження)

Формування системи збуту

№ п/п	Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
	коду, необхідність комерційної підтримки, пріоритет безпеці над ціною			команда для підтримки процесу продажу
2	Технічні команди: оцінка через документацію, пробне використання перед покупкою, порівняння альтернатив на основі технічних характеристик, бюджет обмежений	Надання повної документації та прикладів, наявність безкоштовної мінімальної версії, технічна підтримка	Канал нульового рівня	Автоматизований продаж через платформи, частково власна підтримка для преміум-клієнтів
3	Індивідуальні користувачі: висока чутливість до ціни, оцінка через документацію та приклади, пріоритет на простоті використання	Доступна документація та приклади, підтримка через публічні канали, наявність безкоштовної мінімальної версії	Канал нульового рівня	Автоматизований продаж через платформи

Таблиця 4.22

Концепція маркетингових комунікацій

№ п/п	Специфіка поведінки цільових клієнтів	Канали комуні кацій, якими користуються цільові клієнти	Ключові пози ції, обрані для позиціонування	Завдання рек ламного пові домлення	Концепція рек ламного звернення
1	Ретельна оцінка безпеки та надійності рішень перед впровадженням	Галузеві конференції LinkedIn, спеціалізовані платформи	Безпека Signal, комерційна підтримка	Презентація надійності та відповідності стандартам безпеки	Демонстрація безпечної альтернативи для корпоративної автоматизації
2	Пошук ефективних технічних рішень через документацію	GitHub, Reddit, Stack Overflow, технічні блоги	Зручність API, безпека Signal, якість документації	Показати простоту інтеграції та технічні переваги	Технічні інструкції та приклади реального використання
3	Швидке прийняття рішень на основі доступності інформації	YouTube, GitHub, технічні форуми, Discord	Простота використання, безпека	Зацікавити унікальними можливостями продукту	Відеодемонстрації та швидкий старт з мінімальним порогом входу

ВИСНОВОК ДО ЧЕТВЕРТОГО РОЗДІЛУ

У межах даного розділу було проведено комплексний аналіз ринкових та технічних аспектів для визначення комерційного потенціалу розробленої JavaScript бібліотеки. Проведене дослідження підтверджує можливість ринкової комерціалізації проєкту. Існує чіткий попит, зумовлений глобальним підвищенням потреби у приватності комунікацій і стрімким зростанням популярності чат-ботів. Оціночна рентабельність у суміжних галузях вказує на економічну привабливість ніші.

Щодо перспектив впровадження, стан конкуренції є сприятливим, оскільки прямі аналоги не пропонують додаткового функціоналу, як-от черг повідомлень чи інструментів для реалізації покрокових сценаріїв. Конкурентоспроможність проєкту ґрунтується саме на цій унікальній функціональності, зручності використання та фокусі на безпеці платформи Signal. Однак впровадження ускладнюється високими бар'єрами входження, зокрема технічною складністю протоколу, та високою залежністю від політики Signal Foundation.

Враховуючи ці фактори, з трьох розглянутих варіантів ринкового впровадження, незважаючи на привабливість стратегії з надання послуг із розробки асистентів, було обрано розповсюдження бібліотеки як платного програмного продукту з різними рівнями ліцензування в залежності від бажаної кількості розширень і типом застосування.

Отже, подальша імплементація проєкту є доцільною. Аналіз підтвердив, що проєкт вирішує реальну проблему, яка не закрита існуючими інструментами. Обрана стратегія впровадження мінімізує початкові витрати та дозволяє органічно розвивати продукт, спираючись на реальні потреби ринку.

ВИСНОВКИ

У магістерській роботі було вирішено актуальне завдання із забезпечення автоматизації взаємодії із користувачами у захищеному середовищі месенджера Signal шляхом створення спеціалізованого програмного забезпечення.

У ході виконання роботи було досліджено модель безпеки платформи обміну Signal, її філософію щодо надання послуг (обов'язкове наскрізне шифрування за замовчуванням, відсутність історії на сервері) та встановлено їх вплив на відсутність офіційного API для сторонніх розробників.

Оскільки все програмне забезпечення платформи знаходиться у відкритому доступі, це зумовило розвиток неофіційних інструментів та бібліотек від спільноти, зокрема Signal-cli, який має на меті надати інтерфейси для інтеграції месенджера у робочі процеси без потреби в графічному інтерфейсі.

За результатами огляду існуючих бібліотек для створення персональних асистентів, зокрема Signalbot, Signal-bot, Signal-cli-rest-api та Signal-rest-ts, що побудовані на цьому рішенні, було встановлено, що всі вони мають досить обмежений функціонал і є складними в налаштуванні.

На основі проведеного огляду, а також дослідження наукових напрацювань щодо визначень сучасних діалогових помічників, було упорядковано вимоги до бібліотеки, очікувану функціональність та визначено межі її відповідальності, а також розглянуто засоби та алгоритми, що лежатимуть в основі рішення.

Під час розроблення бібліотеки основний акцент було зроблено на процесі отримання та обробки вхідних даних, їх безпечному збереженні та забезпеченні розробника функціоналом, що дозволяє визначити метод формування відповіді на основі правил: співставлення тексту за командами, за ключовими словами або регулярними виразами, а також за типом вхідної події.

Головним здобутком роботи є впровадження стратегії для управління розмовою на основі моделі скінченних автоматів, до якої для більшої гнучкості взаємодії з користувачем у середовищі платформи обміну повідомленнями було

додано врахування попереднього контексту розмови, можливості визначити обмеження тривалості діалогу чи окремого його етапу, встановити правила, що спрацьовують незалежно від поточного стану автомата.

Для підтвердження працездатності та можливості практичного застосування рішення було проведено модульне, інтеграційне та наскрізне тестування найсуттєвіших для роботи персональних асистентів функцій бібліотеки. Зокрема, наскрізне тестування було проведено з використанням двох діалогових помічників, що взаємодіяли між собою у месенджері Signal.

На останньому етапі роботи було розроблено стартап-проєкт і проведено маркетинговий аналіз, за яким було встановлено стратегію виходу на ринок через модель комерційного ліцензування, ціна якої залежатиме від використання у приватній чи продуктовій сфері.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Chatbot Market Size and Share Analysis - Growth Trends and Forecasts [Електронний ресурс] // Mordor Intelligence. – 2025. – Режим доступу: https://www.mordorintelligence.com/industry-reports/global-chatbot-market.
2. Bogos C. E. A security analysis comparison between Signal, WhatsApp and Telegram [Електронний ресурс] / C. E. Bogos, R. Mocanu, E. Simion. – 2023. – Режим доступу: https://eprint.iacr.org/2023/071.pdf.
3. Secure Messaging Apps Comparison [Електронний ресурс] // Secure Messaging Apps. – Режим доступу: https://www.securemessagingapps.com/.
4. Curry D. Signal Revenue and Usage Statistics [Електронний ресурс] / David Curry // Business of Apps. – 2025. – Режим доступу: https://www.businessofapps.com/data/signal-statistics/.
5. Signal Users by Country 2024 [Електронний ресурс] // World Population Review. – 2025. – Режим доступу: https://worldpopulationreview.com/country-rankings/signal-users-by-country.
6. Signal. Signal – офіційний репозиторій [Електронний ресурс] // GitHub. – Режим доступу: https://github.com/signalapp.
7. Schroder S. When SIGNAL hits the Fan [Електронний ресурс] / S. Schroder, M. Huber, D. Wind, C. Rottermanner // NDSS Symposium. – 2017. – Режим доступу: https://www.ndss-symposium.org/wp-content/uploads/2017/09/09-when-signal-hits-the-fan-on-the-usability-and-security-of-state-of-the-art-secure-mobile-messaging.pdf.
8. Andries S. A survey on the security protocols employed by mobile messaging applications [Електронний ресурс] / S. Andries, A. D. Miron, A. Cristian, E. Simion. – 2022. – Режим доступу: https://eprint.iacr.org/2022/088.
9. Whittaker M. Privacy is Priceless, but Signal is Expensive [Електронний ресурс] / M. Whittaker, J. Lund // Signal Blog. – 2023. – Режим доступу: https://signal.org/blog/signal-is-expensive/.

10. AsamK. signal-cli: Unofficial commandline interface for Signal [Електронний ресурс] // GitHub. – Режим доступу: https://github.com/AsamK/signal-cli.
11. McTear M. Spoken Dialogue Technology: Toward The Conversational User Interface [Електронний ресурс] / Michael McTear // Academia.edu. – 2004. – Режим доступу: https://www.academia.edu/4730903/Spoken_dialogue_technology_enabling_the_conversational_user_interface.
12. Moussiades L. Chatbots: History, technology, and applications [Електронний ресурс] / L. Moussiades // Academia.edu. – 2020. – Режим доступу: https://www.academia.edu/110291755/Chatbots_History_technology_and_applications.
13. Filip R. signalbot: Python package for creating Signal bots [Електронний ресурс] // GitHub. – Режим доступу: https://github.com/filipre/signalbot.
14. Redis: The Real-time Data Platform [Електронний ресурс]. – Режим доступу: https://redis.io/.
15. Sidneys1. signal-bot: Python module for building Signal bots [Електронний ресурс] // GitHub. – Режим доступу: https://github.com/Sidneys1/signal-bot.
16. Bernhard B. signal-cli-rest-api: Signal Messenger REST API [Електронний ресурс] // GitHub. – Режим доступу: https://github.com/bbernhard/signal-cli-rest-api?tab=readme-ov-file.
17. Pseudogeneric. signal-rest-ts: TypeScript wrapper around signal-cli-rest-api [Електронний ресурс] // GitHub. – Режим доступу: https://github.com/pseudogeneric/signal-rest-ts.
18. libsignal-service-java: A Java library for communicating via Signal [Електронний ресурс] // GitHub. – Режим доступу: https://github.com/signalapp/libsignal-service-java.
19. libsignal: Home to the Signal Protocol [Електронний ресурс] // GitHub. – Режим доступу: https://github.com/signalapp/libsignal.
20. Winston: A logger for just about everything [Електронний ресурс] // GitHub. – Режим доступу: https://github.com/winstonjs/winston.

21. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design / Robert C. Martin. – Upper Saddle River, NJ : Prentice Hall, 2017.
22. Keyv: Simple key-value storage with support for multiple backends [Електронний ресурс] – Режим доступу: https://keyv.org/.
23. Zod: TypeScript-first validation library [Електронний ресурс]. – Режим доступу: https://zod.dev/.
24. Jest: Delightful JavaScript Testing [Електронний ресурс]. – Режим доступу: https://jestjs.io/.
25. Fowler M. Inversion of Control [Електронний ресурс] / Martin Fowler // MartinFowler.com. – 2005. – Режим доступу: https://martinfowler.com/bliki/InversionOfControl.html.
26. Richards M. Fundamentals of Software Architecture: An Engineering Approach [Електронний ресурс] / M. Richards, N. Ford. – O'Reilly Media, 2020. – Режим доступу: https://mrce.in/ebooks/Software-Fundamentals%20of%20Software%20Architecture.pdf.
27. Kani N. Lecture 15: Directed Graphs [Електронний ресурс] / Nickvash Kani // ECE-374-B Archive. – 2024. – Режим доступу: https://ecealgo.com/materials/lecture_slides/lec15.pdf.
28. Brabra H. Dialogue Management in Conversational Systems: A Review of Approaches, Challenges, and Opportunities [Електронний ресурс] / H. Brabra, M. Baez, B. Benatallah. – 2021. – Режим доступу: https://marcosbaez.com/assets/pdf/brabra2021dialogue.pdf.
29. Russo V. Graph-based representations of clarification strategies supporting automatic dialogue management [Електронний ресурс] / V. Russo, A. Mancini, M. Grazioso, M. Di Bratto // Italian Journal of Computational Linguistics. – 2022. – Режим доступу: https://journals.openedition.org/ijcol/984.

ДОДАТОК А

Спеціалізована JavaScript бібліотека для створення персональних асистентів у месенджері Signal

Частина програмного коду персонального асистента

Аркушів: 19

Київ – 2025 рік

Файл «index.js»

```
/*
    Personal Signal messenger assistant that provides password management with
Bitwarden
*/

import * as SiBLi from "../../src/index.js";
import {
  BOT_ACCOUNT,
  SESSION_TTL,
  REDIS_URL,
  RECEIVER_OPTIONS,
  VAULT_PASSWORD,
  VAULT_URL,
} from "../config.js";
import { vaultService } from "../services/VaultService.js";
import { generatePasswordFlow } from "../flows/generatePassword.js";
import { createFolderFlow } from "../flows/folderFlow.js";
import {
  addCredentialFlow,
  editCredentialFlow,
  manageCredentialFlow,
} from "../flows/credentialsFlow.js";
import { addFlowRoutes } from "../routes/flowRoutes.js";
import { addRoutes } from "../routes/routes.js";
import logger from "../../src/logging/index.js";

const {
  SignalClient,
  ContextPlugin,
  MessagingPlugin,
  MiddlewarePlugin,
  SessionPlugin,
  FlowPlugin,
} = SiBLi;

export class SignalVaultBot {
  constructor() {
    this.account = BOT_ACCOUNT;
    this.sessionTtl = SESSION_TTL;
    this.redisUrl = REDIS_URL;
    this.receiverOptions = RECEIVER_OPTIONS;

    vaultService.initialize({
      url: VAULT_URL,
      password: VAULT_PASSWORD,
    });

    this.client = new SignalClient();
    this.client.use(new ContextPlugin());
    this.client.use(new MessagingPlugin());
    this.client.use(new MiddlewarePlugin());
    this.client.use(new SessionPlugin());
```

```
    this.client.use(new FlowPlugin());
  }

  async init() {
    await this.client.initialize();

    this.contextPlugin = this.client.getPlugin("context");
    this.messagingPlugin = this.client.getPlugin("messaging");
    this.middlewarePlugin = this.client.getPlugin("middleware");
    this.sessionPlugin = this.client.getPlugin("session");
    this.flowPlugin = this.client.getPlugin("flow");

    this.receiver = this.messagingPlugin.getReceiver(this.account,
this.receiverOptions);
    this.sessionManager = this.createSessionManager();
    this.router = this.middlewarePlugin.createRouter({ commandPrefix: "/" });

    this.addFlows();
    this.useMiddleware();
    this.addRoutes();
  }

  addFlows() {
    this.client.flow.register(generatePasswordFlow().define());
    this.client.flow.register(createFolderFlow().define());
    this.client.flow.register(addCredentialFlow().define());
    this.client.flow.register(editCredentialFlow().define());
    this.client.flow.register(manageCredentialFlow().define());
  }

  useMiddleware() {
    this.router.use(this.sessionManager.middleware());
    this.router.use(this.client.flow.middleware());
  }

  addRoutes() {
    addFlowRoutes(this.router, {
      flowPlugin: this.flowPlugin,
    });

    addRoutes(this.router, {
      client: this.client,
      account: this.account,
    });
  }

  createSessionManager() {
    const store = this.client.createKeyvSessionStore({
      connection: this.redisUrl,
      logger,
    });

    return this.client.getSessionManager(this.account, {
      ttl: this.sessionTtl,
      logger,
```

```
      strategy: store,
    });
  }

  async start() {
    if (!this.router) {
      await this.init();
    }

    const composer = this.router.compile();

    await this.client.accounts.setProfileAvatar(this.account,
"./picts/avatar.jpg");
    await this.client.accounts.updateProfile(this.account, {
      name: "Personal",
      familyName: "Assistant",
      aboutText: "I can manage your passwords",
    });

    await this.receiver.initialize();

    this.receiver.on("messages", async (messages) => {
      for (const rawMessage of messages) {
        try {
          const ctx = this.contextPlugin.createContext(rawMessage);
          await composer.process(ctx);
        } catch (error) {
          logger.error("Message processing error:", {
            error: error?.message ?? error,
          });
        }
      }
    });

    this.receiver.on("error", (error) => {
      logger.error("Receiver error:", {
        error: error?.message ?? error,
      });
    });

    await this.receiver.listen(this.account);
  }

  async stop() {
    if (this.receiver) {
      await this.receiver.stop();
    }

    if (this.client.plugins) {
      await this.client.plugins.destroyAll();
    }
  }
}
```

Файл «routes.js»

```
import logger from "../../src/logging/index.js";
import { vaultService } from "../services/VaultService.js";
import { verifyUser, checkGroupSecurity, formatStatus } from
"../utils/utils.js";

export function addRoutes(router, { client, account }) {
  const invite =

/\b(?:join|group|accept|invitation)\b[^\s]*(?:\s+)?(https:\/\/signal\.group\/#[^
\s]+)/i;

  router.hears([invite]).do(async (ctx) => {
    const link = ctx.text.match(invite)?.[1];

    const isVerified = await verifyUser(client, account, ctx);

    if (isVerified) {
      try {
        await client.joinGroup(account, link);
        await ctx.reply("Joined your group!");
      } catch (error) {
        await ctx.reply("Failed to join group");
        logger.error("Failed to join group via invite link", {
          error: error?.message ?? error,
          sender: ctx.sender.uuid,
          link,
        });
      }
    } else {
      await ctx.reply("You are not my friend...");
    }
  });

  router.command("unlock").do(async (ctx) => {
    const { isSecure, groupName } = await checkGroupSecurity(client, account,
ctx);
    if (!isSecure) {
      await ctx.replyPrivate(
        `Group "${groupName}" is not secure enough. Anyone can add members to
this group.`
      );
      return;
    }

    try {
      const result = await vaultService.unlock();
      ctx.session.vault = {
        unlockedAt: Date.now(),
        expiresIn: result.expiresIn,
      };

      await ctx.reply("Vault unlocked successfully.");
```

```
  } catch (error) {
    await ctx.reply("Unable to unlock vault. Check logs for details.");
    logger.error("Vault unlock failed", {
      error: error?.message ?? error,
      sender: ctx.sender.uuid,
    });
  }
});

router
  .command("lock")
  .and((ctx) => ctx.session?.vault)
  .do(async (ctx) => {
    try {
      await vaultService.lock();
      delete ctx.session.vault;

      await ctx.reply("Vault locked successfully.");
    } catch (error) {
      await ctx.reply("Unable to lock vault. Check logs for details.");
      logger.error("Vault lock failed.", {
        error: error?.message ?? error,
        sender: ctx.sender.uuid,
      });
    }
  });

router.command("status").do(async (ctx) => {
  const isVerified = await verifyUser(client, account, ctx);

  if (!isVerified) {
    await ctx.reply("You are not my friend :(");
    return;
  }

  try {
    const status = await vaultService.getStatus();
    const statusMessage = formatStatus(status);

    await ctx.replyPrivate(statusMessage);
  } catch (error) {
    await ctx.reply("Failed to get status. Check logs for details.");
    logger.error("Failed to get vault status", {
      error: error?.message ?? error,
      sender: ctx.sender.uuid,
    });
  }
});

router.command("help").do(async (ctx) => {
  const help =
    `Available actions:\n\n` +
    `▶ /unlock - Unlock vault\n` +
    `▶ /lock - Lock vault\n` +
    `▶ /generate - Generate secure password\n` +
```

```
      `► /folder [add 'name'|list|edit|delete|get] - Manage vault folders\n` +
      `► cred/credentials [add|edit|get|delete] - Manage credentials in
folders\n` +
      `► /status - Check vault server status\n` +
      `I can also accept group invites, if you are my buddy.`;
    await ctx.reply(help);
  });
}
```

Файл «flowRoutes.js»

```
import logger from "../../src/logging/index.js";
import { vaultService } from "../services/VaultService.js";
import { formatFolderList, getFoldersOrNull } from "../utils/utils.js";

const hasVault = (ctx) => ctx.session?.vault;

const getFoldersOrReply = async (ctx) => {
  try {
    const folders = await getFoldersOrNull(vaultService);

    if (!folders) {
      await ctx.reply("No folders found in your vault.");
      return null;
    }

    return folders;
  } catch (error) {
    await ctx.reply(`Failed to load folders. Check logs for details.`);
    logger.error("Failed to start password generation", {
      error: error?.message ?? error,
      sender: ctx.sender.uuid,
    });
    return null;
  }
};

export function addFlowRoutes(router, { flowPlugin }) {
  router
    .command("generate")
    .and(hasVault)
    .do(async (ctx) => {
      try {
        await flowPlugin.startFlow(ctx, "generate_password");
      } catch (error) {
        await ctx.reply("Failed to start password generation. Check logs for
details.");
        logger.error("Failed to start password generation", {
          error: error?.message ?? error,
          sender: ctx.sender.uuid,
        });
      }
    });

  router
```

```
    .hears([/\bcred(?:ential)?s?\b\s*add\b/i])
    .and(hasVault)
    .do(async (ctx) => {
      try {
        const folders = await getFoldersOrReply(ctx);
        if (!folders) return;

        await flowPlugin.startFlow(ctx, "credentials_add", { data: { folders }
});
      } catch (error) {
        await ctx.reply("Failed to start add credential flow. Check logs for
details.");
        logger.error("Failed to start add credential flow", {
          error: error?.message ?? error,
          sender: ctx.sender.uuid,
        });
      }
    });

  router
    .hears([/\bcred(?:ential)?s?\b\s*edit\b/i])
    .and(hasVault)
    .do(async (ctx) => {
      try {
        const folders = await getFoldersOrReply(ctx);
        if (!folders) return;

        await flowPlugin.startFlow(ctx, "credentials_edit", { data: { folders }
});
      } catch (error) {
        await ctx.reply("Failed to start edit credential flow. Check logs for
details.");
        logger.error("Failed to start edit credential flow", {
          error: error?.message ?? error,
          sender: ctx.sender.uuid,
        });
      }
    });

  router
    .hears([/\bcred(?:ential)?s?\b\s*(get|delete)\b/i])
    .and(hasVault)
    .do(async (ctx) => {
      try {
        const params = ctx.text.toLowerCase();

        if (params.includes("get") && params.includes("delete")) {
          await ctx.reply("Please choose only one operation: get or delete.");
          return;
        }

        const folders = await getFoldersOrReply(ctx);
        if (!folders) return;

        await flowPlugin.startFlow(ctx, "credentials_manage", {
```

```
          data: { folders, params },
        });
      } catch (error) {
        await ctx.reply("Failed to start credential flow. Check logs for
details.");
        logger.error("Failed to start credential flow", {
          error: error?.message ?? error,
          sender: ctx.sender.uuid,
        });
      }
    });

  router
    .command("folder")
    .and(hasVault)
    .do(async (ctx) => {
      try {
        const args = ctx.match?.args || [];
        const action = args[0]?.toLowerCase();

        if (!action) {
          await ctx.reply("Usage: /folder [add 'name'| list | edit | delete |
get]");
          return;
        }

        if (action === "add") {
          const folderName = args.slice(1).join(" ").trim();

          if (!folderName) {
            await ctx.reply("Please provide a folder name. Example: /folder add
Projects");
            return;
          }

          try {
            const folder = await vaultService.createFolder(folderName);
            await ctx.reply(`Folder "${folder.name}" created (id:
${folder.id}).`);
          } catch (error) {
            await ctx.reply(`Failed to add folder. Check logs for details.`);

            logger.error("Failed to add folder", {
              error: error?.message ?? error,
              sender: ctx.sender.uuid,
            });
          }
          return;
        }

        const needsFolders = ["list", "edit", "delete", "get"];

        if (!needsFolders.includes(action)) {
          await ctx.reply("Usage: /folder [add 'name'| list | edit | delete |
get]");
```

```
          return;
        }

        const folders = await getFoldersOrReply(ctx);
        if (!folders) return;

        if (action === "list") {
          const folderList = formatFolderList(folders);
          await ctx.reply(`Your folders:\n${folderList}`);
        } else {
          await flowPlugin.startFlow(ctx, "folder_details", {
            data: { params: action, folders },
          });
        }
      } catch (error) {
        await ctx.reply("Folder command failed. Check logs for details.");
        logger.error("Folder command failed", {
          error: error?.message ?? error,
          sender: ctx.sender.uuid,
        });
      }
    });
}
```

Файл «utils.js»

```
import logger from "../../src/logging/index.js";

export async function verifyUser(client, account, ctx) {
  try {
    const contacts = await client.listContacts(account);
    return contacts.some((contact) => contact.uuid === ctx.sender.uuid);
  } catch (error) {
    logger.error("Failed to verify user", { error: error.message ?? error });
    return false;
  }
}

export async function checkGroupSecurity(client, account, ctx) {
  if (!ctx.isFromGroup()) {
    return { isSecure: true };
  }

  try {
    const groups = await client.listGroups(account);
    const fromGroup = groups.find((g) => g.id === ctx.group.id);

    if (!fromGroup) {
      return { isSecure: false, groupName: "unknown group" };
    }

    const isSecure = fromGroup.permissionAddMember !== "EVERY_MEMBER";
    return { isSecure, groupName: fromGroup.name || "unnamed group" };
  } catch (error) {
    logger.error("Failed to check group security", { error: error.message ??
```

```
error });
    return { isSecure: false, groupName: "unknown group" };
  }
}

export async function getFoldersOrNull(vaultService) {
  const folders = await vaultService.listFolders();
  return folders.length === 0 ? null : folders;
}

export function formatFolderList(folders) {
  if (!folders || folders.length === 0) {
    return "No folders found";
  }

  return folders
    .map((folder, index) => `${index + 1}. ${folder.name} (id: ${folder.id})`)
    .join("\n");
}

export function formatCredentialList(credentials) {
  if (!credentials || credentials.length === 0) {
    return "No credentials found";
  }

  return credentials
    .map((cred, index) => {
      const username = cred.login?.username ?? "";
      return `${index + 1}. ${cred.name}${username ? ` (${username})` : ""} [id:
${cred.id}]`;
    })
    .join("\n");
}

export function formatStatus(status) {
  return (
    `Vault Status:\n\n` +
    `Server: ${status.serverUrl}\n` +
    `User: ${status.userEmail}\n` +
    `Status: ${status.status}\n` +
    `User ID: ${status.userId}\n` +
    `Last Sync: ${status.lastSync}`
  );
}
```

Файл «folderFlow.js»

```
import logger from "../../src/logging/index.js";
import { Flow } from "../../src/plugins/index.js";
import { vaultService } from "../services/VaultService.js";
import { formatFolderList } from "../utils/utils.js";

export function createFolderFlow() {
  return new Flow("folder_details")
```

```
    .ttl(3 * 60 * 1000)

    .interrupt("/cancel", async ({ ctx, transition }) => {
      await ctx.reply("Scenario is cancelled.");
      await transition.abort("user_interrupt");
    })

    .node("ask_folder_id", (node) =>
      node
        .initial()
        .enter(async ({ ctx, flow }) => {
          const folders = flow.data.folders;
          const folderList = formatFolderList(folders);

          await ctx.reply(`Please choose folder id:\n\n${folderList}`);
        })

        .onInput(async ({ ctx, flow, transition }) => {
          const visits = flow.state.history.filter((nodeId) => nodeId ===
"ask_folder_id").length;
          if (visits >= 2) {
            await ctx.reply("Too many invalid attempts. Stopping scenario.");
            await transition.abort("too_many_attempts");
            return;
          }

          const folderId = ctx.text.trim();
          const folderExists = flow.data.folders.some((folder) => folder.id ===
folderId);

          if (!folderExists) {
            await ctx.reply("Folder ID not found. Please check the list and try
again.");
            await transition.repeat();
            return;
          }

          flow.data.folderId = folderId;
          await transition.branch();
        })

        .branch([
          { when: ({ flow }) => flow.data.params === "edit", to: "ask_new_name"
},
          { when: ({ flow }) => flow.data.params === "delete", to:
"delete_folder" },
          { when: ({ flow }) => flow.data.params === "get", to: "get_folder" },
        ])
    )

    .node("ask_new_name", (node) =>
      node
        .enter(async ({ ctx }) => {
          await ctx.reply("Enter new folder name:");
        })
```

```
        .onInput(async ({ ctx, flow, transition }) => {
          const newName = ctx.text.trim();

          if (!newName) {
            await ctx.reply("Folder name cannot be empty.");
            await transition.repeat();
            return;
          }

          flow.data.newName = newName;
          await transition.next();
        })

        .next("edit_folder")
    )

    .node("edit_folder", (node) =>
      node
        .enter(async ({ ctx, flow }) => {
          const { folderId, newName } = flow.data;

          try {
            const result = await vaultService.updateFolder(folderId, newName);

            await ctx.reply(
              `Folder renamed successfully:\nID: ${result.id}\nNew name:
${result.name}`
            );
          } catch (error) {
            await ctx.reply(`Failed to edit folder. Check logs for details.`);

            logger.error("Failed to edit folder", {
              error: error?.message ?? error,
              sender: ctx.sender.uuid,
            });
          }
        })
        .complete()
    )

    .node("delete_folder", (node) =>
      node
        .enter(async ({ ctx, flow }) => {
          const { folderId } = flow.data;

          try {
            await vaultService.deleteFolder(folderId);

            await ctx.reply(`Folder deleted successfully (ID: ${folderId})`);
          } catch (error) {
            await ctx.reply(`Failed to delete folder. Check logs for details.`);

            logger.error("Failed to delete folder", {
              error: error?.message ?? error,
```

```
              sender: ctx.sender.uuid,
            });
          }
        })
        .complete()
    )

    .node("get_folder", (node) =>
      node
        .enter(async ({ ctx, flow }) => {
          const { folderId } = flow.data;

          try {
            const result = await vaultService.getFolder(folderId);

            await ctx.reply(`Folder details:\nID: ${result.id}\nName:
${result.name}`);
          } catch (error) {
            await ctx.reply(`Failed to get folder. Check logs for details.`);

            logger.error("Failed to get folder", {
              error: error?.message ?? error,
              sender: ctx.sender.uuid,
            });
          }
        })
        .complete()
    );
}
```

Файл «generatePassword.js»

```
import { Flow } from "../../src/plugins/index.js";
import { vaultService } from "../services/VaultService.js";
import logger from "../../src/logging/index.js";

const FIELD_CONFIG = {
  length: {
    prompt: "Enter desired password length (8-64):",
    validate: (raw) => {
      const value = Number.parseInt(raw, 10);

      if (Number.isNaN(value)) {
        return { valid: false, error: "Please enter a number." };
      }

      if (value < 8 || value > 64) {
        return { valid: false, error: "Length must be between 8 and 64." };
      }

      return { valid: true, value };
    },
    format: (value) => `${value}`,
  },
};
```

```
function formatSettings(settings) {
  const entries = Object.entries(settings || {});

  return entries
    .filter(([key]) => FIELD_LABELS[key])
    .map(([key, value]) => {
      const label = FIELD_LABELS[key];
      const rendered = typeof value === "boolean" ? (value ? "yes" : "no") :
value;
      return `- ${label}: ${rendered}`;
    })
    .join("\n");
}

export function generatePasswordFlow() {
  return new Flow("generate_password")
    .ttl(2 * 60 * 1000)

    .metadata({
      description: "Generate password with custom settings",
      tags: ["password", "generation"],
    })

    .onEnter(async ({ ctx, flow }) => {
      flow.data.settings = flow.data.settings ?? { ...DEFAULT_SETTINGS };
      await ctx.reply("Welcome! Let's generate your new password.");

      logger.info("Password generation flow started", {
        user: ctx.sender.uuid,
      });
    })

    .onError(async ({ ctx, transition }) => {
      await ctx.reply("Flow timeout reached.");
      await transition.abort("timeout");
    })

    .interrupt("/cancel", async ({ ctx, transition }) => {
      await ctx.reply("Password generation cancelled.");
      await transition.abort("user_interrupt");
    })

    .node("ask_mode", (node) =>
      node
        .initial()
        .enter(async ({ ctx }) => {
          await ctx.reply("Do you want to use default settings or custom?");
        })

        .onInput(async ({ ctx, flow, transition }) => {
          const choice = cleanInput(ctx.text);

          if (choice === MODES.DEFAULT) {
            flow.data.mode = MODES.DEFAULT;
```

```
            flow.data.settings = { ...DEFAULT_SETTINGS };
            await transition.branch();
            return;
          }

          if (choice === MODES.CUSTOM) {
            flow.data.mode = MODES.CUSTOM;
            flow.data.settings = { ...DEFAULT_SETTINGS };
            await transition.branch();
            return;
          }

          await ctx.reply("Please choose 'default' or 'custom'.");
          await transition.repeat();
        })

        .branch([
          { when: ({ flow }) => flow.data.mode === MODES.DEFAULT, to: "get_pass"
},
          { when: ({ flow }) => flow.data.mode === MODES.CUSTOM, to: "menu" },
        ])
    )
    .node("menu", (node) =>
      node
        .enter(async ({ ctx, flow }) => {
          const settings = flow.data.settings ?? { ...DEFAULT_SETTINGS };
          flow.data.settings = settings;

          let menu =
            "Choose what to customize or type 'done':\n\n" +
            "length  - Set password length\n" +
            "special - Include special characters\n" +
            "numbers - Include numbers";

          menu += `\n\nCurrent settings:\n${formatSettings(settings)}`;
          await ctx.reply(menu);
        })

        .onInput(async ({ ctx, flow, transition }) => {
          const choice = cleanInput(ctx.text);

          if (choice === "length") {
            flow.data.currentField = "length";
          }

          if (choice === "done" || choice === "length") {
            await transition.branch();
            return;
          }

          const settings = flow.data.settings ?? { ...DEFAULT_SETTINGS };
          const toggleField = TOGGLE_FIELDS[choice];

          if (toggleField) {
            settings[toggleField] = !settings[toggleField];
```

```
            flow.data.settings = settings;

            await ctx.reply(
              `${FIELD_LABELS[toggleField]}: ${settings[toggleField] ? "enabled"
: "disabled"}`
            );
            await transition.repeat();
            return;
          }

          await ctx.reply("Type 'length', 'special', 'numbers', or 'done'.");
          await transition.repeat();
        })

        .branch([
          { when: ({ ctx }) => cleanInput(ctx.text) === "done", to: "get_pass"
},
          { when: ({ ctx }) => cleanInput(ctx.text) === "length", to:
"set_value" },
        ])
    )

    .node("set_value", (node) =>
      node
        .enter(async ({ ctx, flow }) => {
          const fieldKey = flow.data.currentField;
          const config = FIELD_CONFIG[fieldKey];
          await ctx.reply(config.prompt);
        })

        .onInput(async ({ ctx, flow, transition }) => {
          const fieldKey = flow.data.currentField;
          const config = FIELD_CONFIG[fieldKey];

          const result = config.validate(ctx.text.trim());

          if (result.valid) {
            flow.data.settings[fieldKey] = result.value;
            const label = FIELD_LABELS[fieldKey];

            await ctx.reply(`${label} set to ${config.format(result.value)}`);
            await transition.next();
            return;
          }

          await ctx.reply(`${result.error} Try again:`);
          await transition.repeat();
        })

        .next("menu")
    )

    .node("get_pass", (node) =>
      node
        .enter(async ({ ctx, flow }) => {
```

```
          const options = { ...DEFAULT_SETTINGS, ...(flow.data.settings || {})
};

          try {
            const password = await vaultService.generatePassword(options);
            flow.data.generatedPassword = password;

            await ctx.reply(`Generated password:\n\n${password}`);
          } catch (error) {
            await ctx.reply(`Failed to generate password. Check details in
logs.`);

            logger.error("Failed to generate password", {
              user: ctx.sender.uuid,
              error: error?.message ?? error,
            });
          }
        })
        .complete()
    );
}
```

Файл «createCredentialFlow.js»

```
import { Flow } from "../../src/plugins/index.js";

export const createCredentialFlow = (id, description) =>
  new Flow(id)
    .ttl(2 * 60 * 1000)
    .metadata({
      description,
      tags: ["credentials", "vault"],
    })
    .onError(async ({ ctx, transition }) => {
      await ctx.reply("Something went wrong. Try again later");
      await transition.abort("error");
    })
    .interrupt("/cancel", async ({ ctx, transition }) => {
      await ctx.reply("Credentials are not saved!");
      await transition.abort("user_interrupt");
    });
```

Файл «credentialsFlow.js»

```
import logger from "../../src/logging/index.js";
import { vaultService } from "../services/VaultService.js";
import { createCredentialFlow } from "./createCredentialFlow.js";
import { formatFolderList } from "../utils/utils.js";

const validateSelection = async ({ ctx, flow, transition }) => {
  const folderId = ctx.text.trim();
  const folderExists = flow.data.folders.some((folder) => folder.id ===
folderId);
```

```
  if (!folderExists) {
    await ctx.reply("Folder ID not found. Please check the list and try
again.");
    await transition.repeat();
    return null;
  }

  flow.data.folderId = folderId;
  return folderId;
};
export function addCredentialFlow() {
  return createCredentialFlow("credentials_add", "Add credential")
    .node("choose_folder", (node) =>
      node
        .initial()

        .enter(async ({ ctx, flow }) => {
          const folderList = formatFolderList(flow.data.folders);
          await ctx.reply(`Choose a folder:\n\n${folderList}`);
        })

        .onInput(async ({ ctx, flow, transition }) => {
          const folderId = await validateSelection({ ctx, flow, transition });
          if (!folderId) return;
          await transition.next();
        })

        .next("add_credential")
    )

    .node("add_credential", (node) =>
      node
        .enter(async ({ ctx }) => {
          await ctx.reply("Enter credential name:");
        })

        .onInput(async ({ ctx, flow, transition }) => {
          const name = ctx.text.trim();

          if (!name) {
            await transition.repeat();
            return;
          }

          flow.data.credentialName = name;
          await transition.next();
        })

        .next("ask_username")
    )
    .node("ask_username", (node) =>
      node
        .enter(async ({ ctx }) => {
          await ctx.reply("Enter username:");
        })
```

```
        .onInput(async ({ ctx, flow, transition }) => {
          flow.data.credentialUsername = ctx.text.trim();
          await transition.next();
        })

        .next("ask_password")
    )
    .node("ask_password", (node) =>
      node
        .enter(async ({ ctx }) => {
          await ctx.reply("Enter password:");
        })

        .onInput(async ({ ctx, flow, transition }) => {
          const password = ctx.text.trim();

          if (!password) {
            await transition.repeat();
            return;
          }

          flow.data.credentialPassword = password;
          await transition.next();
        })

        .next("add")
    )
    .node("add", (node) =>
      node
        .enter(async ({ ctx, flow }) => {
          try {
            const { folderId, credentialName, credentialUsername,
credentialPassword } = flow.data;
            const result = await vaultService.createCredential({
              folderId,
              name: credentialName,
              username: credentialUsername,
              password: credentialPassword,
            });
            await ctx.reply(`Credential "${credentialName}" added (id:
${result.id}).`);
          } catch (error) {
            await ctx.reply(`Failed to add folder. Check logs for details.`);
            logger.error("Failed to add folder to the vault", {
              error: error?.message ?? error,
              sender: ctx.sender.uuid,
            });
          }
        })
        .complete()
    );
}
```