Network Monitor projekt – Fejlesztési fázisok dokumentációja

Hallgató neve: Németh Gyula
Neptun-kód: PLBHOJ
Dokumentum célja: a generatív AI felé megfogalmazott összes érdemi prompt és fejlesztési kérés bemutatása, tanári értékelés támogatására

---

I. fázis – Alapötlet és célmeghatározás

Célok: A projekt alapötletének kidolgozása és a fő célok meghatározása. Ebben a fázisban a fejlesztő definiálta, hogy egy Java nyelven írt hálózatmonitorozó alkalmazást kíván létrehozni grafikus felhasználói felülettel, amely oktatási célokra is felhasználható. A célkitűzés részeként rögzítésre került, hogy az alkalmazás valós hálózati méréseket tudjon végezni (pl. ping és csomagküldési tesztek), miközben a használata szemléletes és interaktív marad a tanulók számára.

MI-hez intézett kérdések (promptok):
- „Network Monitor alkalmazás készítése Java nyelven.”
- „Grafikus felhasználói felület alkalmazása.”

Fejlesztési megközelítés: A fejlesztő a generatív MI bevonásával körvonalazta az alkalmazás architektúráját és funkcióit. Ebben a korai szakaszban iteratív tervezést alkalmazott: először azonosította a legfontosabb komponenseket és követelményeket, majd a MI segítségével készített egy alap projektvázat (fő osztályok, ablakok, menüpontok). A grafikus felület tekintetében a MI javaslatot tett a JavaFX használatára, amelyet a fejlesztő elfogadott. A kezdeti iterációk célja egy működőképes prototípus volt, amely már tartalmazza a fő ablakelemeket és előkészíti a terepet a hálózati funkciókhoz.

Alkalmazott eszközök: Az alkalmazás nyelvéül a Java lett kiválasztva, a grafikus felület megvalósításához pedig a JavaFX keretrendszer. A tervezés során előzetesen szó esett a hálózati funkciókhoz használni tervezett protokollokról is: ICMP a ping műveletekhez és UDP a csomagküldési tesztekhez. Ezekhez a Java standard könyvtárait (például a `java.net` csomag osztályait) tervezték felhasználni. Ebben a fázisban elsősorban tervezési eszközök és a MI által sugallt technológiai stack került rögzítésre (JavaFX a GUI-hoz, Java socket API a hálózati kommunikációhoz).

Nehézségek: A fő kihívás ebben a szakaszban a projekt hatókörének pontos meghatározása volt. A generatív MI kezdeti válaszainak értelmezése és finomítása némi odafigyelést igényelt: például biztosítani kellett, hogy a MI által javasolt megoldások (pl. mely GUI toolkit használata) összhangban legyenek a fejlesztő elképzeléseivel és a projekt oktatási céljaival. Felmerült a technológiai választások kérdése (Swing vagy JavaFX a felülethez), és a MI visszajelzései alapján végül a

modernebb JavaFX mellett döntött a fejlesztő. Emellett figyelni kellett arra is, hogy a kitűzött funkciók (ping, különböző csomagküldési módok) reálisan megvalósíthatók legyenek a rendelkezésre álló idő és erőforrások keretében.

Eredmény: A fázis végére kialakult a Network Monitor projekt koncepciója és elkészült egy kezdeti prototípus. Ez a prototípus már tartalmazta a grafikus felület alapjait (ablak keret, vezérlőelemek helyei) és előkészítette a kódstruktúrát a hálózati funkciók befogadására. A hallgató és a generatív MI együttműködésének köszönhetően megszületett egy világos fejlesztési terv, amely lefektette a további munka alapjait, és az alkalmazás fejlesztése magabiztosan elindulhatott.

II. fázis – Alap hálózati funkciók

Célok: A hálózatdiagnosztikai alapfunkciók implementálása a programban. Ebbe a fázisba tartozik egy egyszerű ICMP ping művelet megvalósítása, valamint egy UDP alapú csomagküldési teszt létrehozása unicast (egy címzett) módon. A cél az volt, hogy a program képes legyen egy megadott távoli host elérhetőségének mérésére (ping), illetve egy tesztcsomag küldésére egy adott IP-címre, és ezáltal alapvető visszajelzést adjon a hálózat működéséről.

MI-hez intézett kérdések (promptok):
- „ICMP ping mérési funkció beépítése."
- „UDP alapú csomagteszt funkció implementálása."
- „Unicast csomagküldési mód támogatása."

Fejlesztési megközelítés: Az alapvető hálózati funkciók hozzáadását a fejlesztő lépésről lépésre végezte a MI támogatásával. Először az ICMP ping funkció került kialakításra: a hallgató rákérdezett a MI-nál, hogyan valósítható meg a ping művelet Java nyelven. A MI javaslatot tett például az `InetAddress.isReachable()` metódus használatára, amellyel egyszerűen le lehet kérdezni, hogy egy host elérhető-e. A kapott kódot a fejlesztő beépítette és tesztelte a saját hálózatán. Ezt követően a fókusz a csomagküldésre terelődött: a MI útmutatásai alapján a hallgató megírta a UDP alapú csomagtesztet, először unicast módon. Minden funkció implementálása után a programot futtatta és ellenőrizte az eredményt, így az esetlegesen felmerülő hibákat azonnal orvosolni tudta a MI további útmutatásai segítségével. Az iteratív fejlesztés tehát ebben a fázisban is jelen volt: a hallgató mindig egy konkrét feladatra (ping, majd UDP küldés) koncentrált, és addig finomította a megoldást a MI-val, amíg az megfelelően nem működött.

Alkalmazott eszközök: Az ICMP ping megvalósításához a Java beépített hálózati eszköztárát alkalmazták, különös tekintettel az `InetAddress` osztályra. Az `InetAddress.getByName("host").isReachable(timeout)` metódus lehetőséget adott egy cél host elérésének tesztelésére. A UDP alapú csomagküldéshez a Java socket API-t vették igénybe: a fejlesztő a MI által sugallt módon a `DatagramSocket` és `DatagramPacket` osztályokat használta egy egyszerű csomag összeállítására és elküldésére. Unicast mód esetén a csomag céljaként egyetlen IP-címet és portot kellett

megadni; ezt a feladatot a program felületi beállításain keresztül a felhasználó határozhatja meg (pl. beviteli mezőkben). Mindezen eszközök a Java Standard Edition részei, így külső könyvtár használatára nem volt szükség ebben a fázisban.

Nehézségek: Ezen funkciók implementálása során néhány technikai kihívás merült fel. Az ICMP ping esetében korlátot jelentett, hogy Java-ban nincs közvetlen, magas szintű API a ping parancs teljes funkcionalitására (mivel az ICMP echo request küldése alacsony szintű művelet). A MI által javasolt `isReachable()` metódus a háttérben próbálkozik ICMP-vel, de sok rendszeren csak adminisztrátori jogosultsággal működik megfelelően, egyébként TCP fallback megoldást használhat. Ezt a hallgatónak tesztelnie kellett a saját környezetében, és szükség esetén alternatív megoldást találni (pl. a rendszer `ping` parancsának meghívását). A UDP csomagküldésnél kihívást jelentett, hogy a küldött csomag fogadására nem feltétlenül volt partner a hálózatban, így a teszt csak annyit jelzett, hogy a csomagot sikerült elküldeni. A MI felhívta a figyelmet a kivételek (pl. `IOException`) kezelésének fontosságára is, hiszen a hálózati műveletek során számolni kell elérhetetlenséggel vagy időtúllépéssel. A hallgató ezeket a tanácsokat megfogadva try-catch blokkokkal látta el a kritikus részeket, és log üzenetekkel jelezte, ha egy host nem érhető el vagy a csomagküldés nem sikerült.

Eredmény: A program képessé vált alapszintű hálózati mérések elvégzésére. A felhasználó a grafikus felületen megadhat egy tetszőleges IP-címet vagy hostnevet, és indíthat egy ping tesztet, amelynek eredményeként a program jelzi, hogy a cél elérhető-e (és tipikusan mennyi idő alatt). Emellett lehetővé vált egy egyszerű UDP csomag elküldése is egy kiválasztott célállomásra unicast módon, ami a későbbi fejlesztések alapját képezi. E fázis végére a Network Monitor rendelkezett azokkal az alapvető funkciókkal, amelyekkel már hasznos hálózati információkat tudott nyújtani, igazolva a projekt alapötletének életképességét.

III. fázis – Haladó hálózati módok

Célok: A csomagküldési funkciók bővítése további hálózati kommunikációs módokkal: a **broadcast**, **multicast** és **anycast** típusú forgalom kezelése. A cél az volt, hogy a Network Monitor ne csak egyetlen címzetthez tudjon üzenetet küldeni, hanem támogassa a hálózaton belüli szélesebb körű kommunikációt is. A broadcast funkció lehetővé teszi, hogy egy csomagot egy adott alhálózat minden eszközének elküldjünk (pl. egy router által meghirdetett broadcast címen keresztül). A multicast mód célja, hogy a program csomagokat tudjon küldeni egy multicast csoportnak (valamint fogadni onnan, ha szükséges), míg az anycast esetében a koncepció bemutatása volt a cél – azaz, hogy a program kezelni tudja azt a helyzetet, amikor egy adott címhez tartozó **bármelyik** elérhető szerver válaszol (anycast hálózati minta).

MI-hez intézett kérdések (promptok):
- „Broadcast csomagküldési mód megvalósítása.”
- „Multicast csomagtesztek támogatása.”
- „Anycast tesztek implementálása.”

Fejlesztési megközelítés: A haladó hálózati módok implementációja modulárisan, egymás után történt a MI iránymutatásával. Először a broadcast funkció került sorra. A MI magyarázatot adott arra, miként lehet egy UDP csomagot broadcast címre küldeni Java-ban: felhívta a figyelmet a `DatagramSocket` beállítására (`setBroadcast(true)`), illetve arra, hogy például a 255.255.255.255 cím használatával a csomag a lokális hálózat minden eszközéhez eljuthat. A fejlesztő implementálta a broadcast küldést, majd tesztelte egy lokális hálózaton, figyelve arra, hogy a csomagot a hálózat eszközei valóban megkapják-e (ez jellemzően tűzfal-beállításoktól is függ). Ezt követően a multicast funkció következett. A MI javasolta a Java `MulticastSocket` osztály használatát, amellyel a program csatlakozhat egy multicast csoportcímhez (pl. 224.0.0.1) és oda küldhet üzeneteket. A fejlesztő ennek megfelelően kialakította a lehetőséget a felhasználó számára egy multicast cím megadására és a csoporthoz tartozó üzenetküldésre. Az anycast esetén a helyzet bonyolultabb volt: mivel az anycast nem egy külön protokoll, hanem hálózati architektúra kérdése, a MI segítséget nyújtott a fogalom értelmezésében és egy lehetséges teszt forgatókönyvben. Végül egy egyszerűsített anycast-szimuláció valósult meg: a program több, azonos szolgáltatást nyújtó szerver IP-címe közül **véletlenszerűen vagy az első válaszoló alapján** választja ki, melyiktől fogad választ, így demonstrálva, hogy nem mindig ugyanattól a címről jön a válasz (ez utal az anycast jellegére). Minden új mód bevezetése után a fejlesztő futtatta a programot és a MI útmutatásai alapján ellenőrizte a működést, így finomhangolva a megoldást (pl. a broadcast flag helyes használatát vagy a multicast csoport elhagyását a program bezárásakor, stb.).

Alkalmazott eszközök: A broadcast küldéshez a Java UDP socket kezelése kibővült azzal a képességgel, hogy broadcast címre is lehessen csomagot küldeni. Ehhez a `DatagramSocket` objektum `setBroadcast(true)` beállítása szükséges, majd a csomagot egy speciális címre (pl. 192.168.1.255 – a helyi hálózat broadcast címe – vagy általánosan 255.255.255.255) kell elküldeni. A multicast funkcióhoz a `MulticastSocket` osztályt alkalmazták, amely lehetővé teszi multicast csoporthoz való csatlakozást (`joinGroup(InetAddress groupAddress)` hívással) és üzenetek küldését egy csoportcímre. A multicast megoldásnál arra is figyelni kellett, hogy a hálózati adapter megfelelően konfigurált legyen (pl. multicast engedélyezése). Az anycast funkcionalitás – mivel nincs rá dedikált Java API – kreatív megoldást igényelt: a program listában tárol több lehetséges cél IP-címet (amik ugyanazt a szolgáltatást kínálják), majd egyesével megpingeli vagy csomagot küld nekik, és amelyiktől először választ kap, azt tekinti "nyertes" anycast célpontnak. Ehhez a már meglévő ping vagy UDP küldési eszközöket használták fel kismértékű módosítással (pl. ciklusban próbálkozás a címekkel). Összességében ebben a fázisban is a Java standard hálózati eszköztárára támaszkodtak (DatagramSocket, MulticastSocket), új konfigurációkkal és logikával kiegészítve.

Nehézségek: A broadcast és multicast funkciók bevezetése során a fejlesztőnek figyelembe kellett vennie a hálózati környezet sajátosságait. Például nem minden hálózat engedi a broadcast csomagok továbbítását, és bizonyos operációs rendszereken vagy tűzfalbeállítások mellett a broadcast üzenetek nem érnek célba. A MI felhívta a figyelmet arra, hogy a teszteléshez érdemes ugyanazon a hálózaton több példányban futtatni az alkalmazást vagy sniffing eszközzel figyelni a forgalmat. A

multicast esetében kihívás volt a megfelelő multicast cím kiválasztása és a csoportkezelés (például, hogy a program kilépéskor leiratkozzon a csoportból, különben a hálózati erőforrások fogva maradhatnak). Az anycast implementációja külön nehézséget jelentett, mivel valós anycastot szimulálni egyetlen gépről nem triviális: a hallgatónak kreatívan kellett eljárnia. A MI ugyan segített ötletekkel (pl. több szerver IP-jének lekérése DNS-ből egy host név alapján), de a megvalósítás limitációit (pl. több internetes szerver pingelése) a projekt keretei között kezelni kellett. Mindenesetre az anycast koncepciót sikerült bemutatni, habár a valós hálózati anycast élmény csak részlegesen modellezhető. Összességében e fázis legfőbb kihívása az volt, hogy a programot felkészítsék a bonyolultabb hálózati kommunikációkra anélkül, hogy a stabilitása sérülne.

Eredmény: A Network Monitor alkalmazás funkcionalitása kibővült a haladó hálózati módokkal. A felhasználó most már választhat a felületen, hogy a csomagküldést milyen módban kívánja futtatni: unicast, broadcast, multicast vagy anycast. Broadcast mód esetén a program egy egész alhálózatnak képes üzenetet küldeni; multicast módban csatlakozni tud egy csoporthoz és oda küld csomagot (vagy akár fogadni is tud, ha ezt is implementálták); anycast módban pedig demonstrálja, hogy több lehetséges cél közül a "legelső" válaszolóhoz kapcsolódik. Ezzel az alkalmazás sokoldalúbbá vált, hiszen lefedi a hálózati kommunikáció különböző eseteit, ami különösen értékes oktatási szempontból: a felhasználók interaktívan megtapasztalhatják a különféle -cast típusú kommunikáció közötti különbségeket.

IV. fázis – Architektúra és stabilitás

Célok: Az alkalmazás architektúrájának továbbfejlesztése a stabil működés és a reszponzív felhasználói élmény érdekében. A korábbi fázisokban kifejlesztett funkciók növelték a program komplexitását, ezért szükségessé vált a **grafikus felület** és a **hálózati műveletek** szétválasztása, illetve a **többszálú futtatás** bevezetése. Az elsődleges cél az volt, hogy a hosszabb ideig tartó vagy folyamatos hálózati tesztek (ping sorozatok, csomagküldési ciklusok) ne akadályozzák a felhasználói felület reagálását – azaz az alkalmazás ne „fagyjon le” mérés közben sem. Emellett felmerült az igény egy **Stop mechanizmus** megtervezésére is, amely lehetővé teszi a felhasználónak a háttérben futó mérések leállítását. Ebben a fázisban tehát az alkalmazás szerkezetének olyan átalakítása volt a cél, ami biztosítja a stabil, jól kezelhető működést a növekvő funkciók mellett.

MI-hez intézett kérdések (promptok):
- „A csomagtesztek ne fagyasszák le a GUI-t, külön szálon fussanak.”
- „A grafikus felület lefagyásának megakadályozása.”

Fejlesztési megközelítés: E fázisban a fejlesztés fő módszere a program **párhuzamosítása** volt. A generatív MI segítségével a fejlesztő átalakította az alkalmazás szerkezetét oly módon, hogy a hosszabb futású hálózati műveletek külön szálon (background thread) fussanak, elkülönítve a JavaFX grafikus felület (UI thread) szálától. A MI javaslatokat tett arra, miként lehet a Java-ban új szálat indítani és kezelni:

például egy `Thread` objektum létrehozásával, vagy a JavaFX `Task` osztályának alkalmazásával a háttérfeladatokhoz. A hallgató először egy egyszerűbb megközelítést választott a MI útmutatásai alapján: minden egyes mérésindítás (pl. ping sorozat) előtt létrehozott egy új szálat, és azon belül hívta meg a hálózati funkciókat, míg a fő szál tovább kezelte a GUI-t. Az iterációk során a fejlesztő többször ellenőrizte a program futását: figyelte, hogy egy mérés elindítása után is tud-e például gombokat nyomni vagy ablakot mozgatni a felületen. Ha mégis tapasztalt akadozást, visszatért a MI-hoz tanácsért. Ennek köszönhetően finomításra került a szálkezelés: bevezetésre kerültek szükség esetén **szinkronizációs mechanizmusok** vagy a JavaFX `Platform.runLater()` hívásai, hogy a háttérszál biztonságosan tudja frissíteni a felületet (például új mérési eredmény megjelenítésekor). A tervezés során előkészítették a Stop funkció alapjait is, bár ennek teljes megvalósítása a következő fázisra maradt. Már itt szó esett arról, hogy egy megosztott flag változó vagy más vezérlési mechanizmus kell a szálak közötti kommunikációhoz, amire a MI adott is javaslatokat.

Alkalmazott eszközök: A többszálú működés megvalósításához a Java beépített **szálkezelési** eszközeit alkalmazták. A programban közvetlenül a `Thread` osztály példányosításával indultak új szálak a háttérfeladatokhoz. (Az egyes mérések logikáját külön `Runnable`-ként vagy lambda kifejezésként adták át a `Thread` konstruktornak.) A JavaFX környezet sajátosságai miatt a grafikus felület módosításait (pl. egy új sor hozzáadása a log ablakhoz vagy a grafikon frissítése) a JavaFX fő szálán kellett végrehajtani – ennek biztosítására a `Platform.runLater()` metódus szolgált, amit szintén bevezettek. Az architekturális módosítások során figyelembe vették a **Java Memory Model** alapelveit is: a MI tanácsai nyomán a fejlesztő úgy tervezte, hogy a később bevezetendő megosztott flag változó (a Stop mechanizmushoz) `volatile` legyen, ezzel garantálva a változó értékének konzisztens látását a különböző szálakon. Mindezek mellett a projekt továbbra is tisztán Java Standard Library eszközöket használt, külön külső függőség nélkül, de az architektúra immár aszinkron, esemény-orientált lett.

Nehézségek: A több szál bevezetése során adódtak kihívások mind a tervezésben, mind a hibakeresésben. Kezdetben a MI által generált példakódot (amely bemutatta, hogyan kell egy új szálat indítani) adaptálni kellett a projekt meglévő kódjához. Ez némi átszervezést igényelt: például a hálózati műveletek kódját ki kellett emelni külön metódusokba vagy osztályokba, hogy azokat a háttérszál könnyen meghívhassa. Az első próbálkozások során előfordult, hogy a GUI még mindig blokkolódott – utólag kiderült, hogy bizonyos hosszabb műveletek (pl. egy nagyobb ciklus vagy várakozás) még mindig a fő szálon futottak. A hallgató ilyenkor visszatért a MI-hoz tanácsért, aki rámutatott az ilyen problémás részekre, így azokat is külön szálba szervezték. Egy másik technikai kihívás a szálak közötti együttműködés biztosítása volt: például annak kezelése, hogy a háttérben gyűlő mérési eredményeket **szálbiztos módon** juttassák el a GUI komponensekhez. Ezt végül a `volatile` kulcsszóval ellátott megosztott változókkal és szükség esetén synchronized blokk használatával oldották meg, a MI iránymutatásait követve. Felmerült a potenciális **erőforrás-szivárgás** kérdése is: a MI jelezte, hogy ügyelni kell a szálak lezárására (pl. ne maradjanak örökké életben feleslegesen), ezért a fejlesztő beiktatott olyan mechanizmust, ami biztosítja, hogy a háttérszál a feladat végeztével valóban befejeződjön. Összességében ebben a fázisban

a program belső működésének megbízhatóságát kellett megalapozni, ami néhány iteráció alatt sikerült is a MI segítségével.

Eredmény: A fejlesztés e szakaszának eredményeként a Network Monitor alkalmazás jóval stabilabb és felhasználóbarátabb lett. A hálózati tesztek futtatása immár nem akasztja meg a felhasználói felületet: a mérésindítás gomb megnyomása után az alkalmazás továbbra is reagál az esetleges további felhasználói műveletekre (pl. ablaktevékenységek), miközben a háttérben zajlik a hálózati kommunikáció. Előkészítésre került továbbá a Stop funkció – ekkorra a program struktúrája már lehetővé tette, hogy egy felhasználói utasítás hatására kulturáltan leálljon egy háttérfolyamat (a megfelelő flag váltásával és szálkezeléssel). Röviden, ez a fázis teremtette meg a program **méretezhetőségének** és **robosztusságának** alapját, lehetővé téve az összetettebb funkciók biztonságos integrálását a következő lépésekben.

V. fázis – Végtelenített mérések

Célok: A hálózati tesztek folyamatos (végtelenített) futtatásának lehetővé tétele, egészen addig, amíg a felhasználó nem kéri azok leállítását. Ennek a fázisnak a célja egy olyan stressztesztfunkció kialakítása volt, amely során a program megszakítás nélkül képes csomagokat küldeni vagy pingelni, és a mérés addig tart, ameddig a felhasználó le nem állítja. Ezzel párhuzamosan biztosítani kellett, hogy a felhasználó kezében legyen az irányítás: egy **Stop gomb** segítségével bármikor véget vethessen a folyamatnak. Fontos cél volt továbbá az alkalmazás **állapotkezelése**, hogy a felhasználó számára is egyértelmű legyen, éppen folyamatban van-e mérés, vagy le van állítva (pl. egy visszajelző felirat vagy eltérő színű gomb jelzi a futó mérést).

MI-hez intézett kérdések (promptok):
- „Csomagtesztek végtelenített futtatása manuális leállításig.”
- „Stop gomb funkció kialakítása.”
- „Java Memory Model kompatibilis szálkezelés alkalmazása.”

Fejlesztési megközelítés: A végtelenített (más szóval folyamatos, ismétlődő) mérések bevezetését a fejlesztő és a MI egy újabb iterációs folyamat során valósította meg. Első lépésként a hallgató módosította a korábbi, fix számú ismétléssel működő ciklusokat egy **végtelen ciklusra**. A MI javaslatot tett egy egyszerű megoldásra: használjanak egy `while` ciklust, amely addig fut, amíg egy `running` nevű logikai flag értéke igaz. Ezt a flag változót a mérés elején igazra állítják, majd a Stop gomb megnyomásakor hamisra; így a ciklus feltétele megszűnik, és a háttérszál kilép. A hallgató ennek megfelelően implementálta a módosítást a ping és csomagküldő rutinokban. A Stop gomb létrehozásakor a MI tanácsokkal látta el: meg kellett oldani, hogy a gomb eseménykezelője elérje a háttérben futó szál `running` változóját. Ezt a fejlesztő úgy valósította meg, hogy a flaget egy, a szál számára is látható helyen (például egy közös objektumban vagy megfelelő hatókörű statikus változóban) tárolta. A MI itt hívta fel a figyelmet a **Java Memory Model** szerinti helyes szálkommunikációra: javasolta a `volatile` kulcsszó használatát a flag deklarációjánál. Ennek hatására a háttérszál

rögtön érzékeli a fő szálról történő változtatást (a Stop gomb megnyomását), és nem fordul elő a tipikus probléma, hogy a cache-elt változó miatt a szál „nem látja” a leállítási kérést. A fejlesztő többször kipróbálta a végtelenített futtatást: először szándékosan figyelte meg, hogy Stop gomb nélkül a program valóban folyamatosan küldi-e a csomagokat (ezzel meggyőződött a végtelen ciklus működéséről). Ezután integrálta a Stop gombot, és tapasztalta, hogy elsőre a szál nem mindig áll le azonnal – ekkor újra a MI segítségét kérte, aki nyomatékosította a `volatile` használatát. A módosítás után a Stop gomb már az elvárásnak megfelelően leállította a háttérfolyamatot. A MI javasolta azt is, hogy egy **biztonsági mechanizmust** építsenek be: például ha a Stop gombot megnyomják, és a szál nem ér véget egy bizonyos időn belül, akkor szakítsa meg (esetleg `thread.interrupt()` hívásával). A hallgató ezeket megfontolta, és mivel a `volatile` flag jól működött, az interrupt hívására nem is volt szükség a végleges megoldásban.

Alkalmazott eszközök: A végtelenített ciklus megvalósításához a Java vezérlési szerkezeteit használták: egy `while(true)` vagy ezzel ekvivalens ciklust építettek be a mérési folyamatba. A ciklus vezérléséhez bevezettek egy megosztott **flag változót** (pl. `volatile boolean running`), amely a ciklus feltételeként szolgált. A Stop funkció a grafikus felületen egy különálló **gomb** formájában jelent meg; ennek eseménykezelő kódja a felhasználó kattintására beállítja a flag változót false értékre. A flag-et `volatile` kulcsszóval deklarálták, biztosítva ezzel, hogy a változó módosítása a fő szálon azonnal látható legyen a háttérszál számára is. Emellett a fejlesztő beépített egy rövid várakozást (pl. `Thread.sleep(100)` milliszekundumos szünet) a végtelen ciklus iterációi közé a MI javaslatára, hogy a folyamatos futás se terhelje le maximálisan a CPU-t, és legyen idő a GUI frissítésére is két üzenet küldése között. A háttérszál leállása után gondoskodni kellett az erőforrások felszabadításáról is, például a nyitva lévő socket bezárásáról – ezt a program a ciklusból kilépve automatikusan elvégezte (a try-with-resources szerkezet vagy a finally blokk révén, amit szintén implementáltak a MI tanácsai alapján).

Nehézségek: A végtelenített mérés bevezetése kapcsán az egyik fő nehézség a program **stabilitásának megőrzése** volt. Egy folyamatosan futó hálózati teszt hosszabb idő után váratlan helyzeteket idézhet elő: például memóriakezelési szempontból figyelni kellett arra, hogy a program ne halmozzon fel végtelen mennyiségű adatot (log vagy grafikon adatok formájában) a végtelen ciklus során. A MI javasolta, hogy korlátozott méretű adatsorozatokat használjanak (pl. a grafikon csak az utolsó N mérési eredményt mutassa), így ezt implementálták a következő fázisban. Technikai kihívást jelentett továbbá a háttérszál **megfelelő leállítása**: kezdetben előfordult, hogy a Stop megnyomása után a szál nem állt le, mert a flag változó értékét a szál nem frissítette időben. Ezt a problémát a `volatile` kulcsszó alkalmazása oldotta meg, de ennek felismerése és tesztelése is időt igényelt. További finomhangolást igényelt, hogy a Stop gombot többször is lehessen használni: például ha a felhasználó újraindít egy mérést leállítás után, a program megfelelően újra tudja indítani a háttérszálat (ehhez resetelni kellett a flaget és kezelni, hogy ne induljon egyszerre két szál – ezt a MI hangsúlyozta). Ezen kihívások mindegyikére sikerült megoldást találni a generatív MI segítségével, biztosítva, hogy a végtelenített teszt funkció megbízhatóan működjön.

Eredmény: A program mostantól képes **folyamatosan**, megszakítás nélkül végezni a hálózati csomagteszteket vagy ping műveleteket, ami lehetővé teszi a tartós terheléses vizsgálatokat és a hálózat stabilitásának hosszú távú megfigyelését. A felhasználói felületen megjelent egy jól látható Stop gomb, amellyel a felhasználó bármikor biztonságosan leállíthatja a futó méréseket. Ennek hatására a háttérszál szabályosan kilép a végtelen ciklusból, és az alkalmazás készen áll új mérések indítására. A Network Monitor ezzel a funkcióval immár egy valódi **stresszteszt** eszközzé is vált, hiszen hosszú ideig képes hálózati forgalmat generálni, és megfigyelni, hogy a hálózat hogyan viselkedik ilyen körülmények között. A Stop mechanizmus beépítésével pedig garantált, hogy a felhasználó mindig kontrollálni tudja a folyamatot, elkerülve a program „elszabadulását” vagy a rendszer túlterhelését.

VI. fázis – Kiterjesztett funkciók

Célok: A projekt funkcióinak kibővítése további hálózati diagnosztikai eszközökkel és mérésekkel, annak érdekében, hogy az alkalmazás átfogóbb képet adhasson a hálózat állapotáról és teljesítményéről. Ebben a fázisban számos új célt határoztak meg: egy **traceroute** funkció implementálását (útvonal-elemzés a célállomás felé vezető úton), a rendszer hálózati kapcsolatait listázó **netstat**-jellegű információk megjelenítését, a **helyi hálózat pásztázását** (aktív eszközök felderítése az adott hálózati szegmensben), valamint hálózati **sebességmérések** megvalósítását (letöltési és feltöltési sebesség meghatározása), és egy **HTTP válaszidő** mérő eszközt. Kiegészítő célként felmerült az **UPnP porttovábbítás** kezelése is, amellyel a program képes lehet automatikusan konfigurálni az útválasztót (routert) bizonyos portok megnyitására a tesztek idejére. Ezekkel a kiterjesztett funkciókkal a Network Monitor egy sokoldalú hálózati szerszámkészletté fejlődhet, alkalmasabbá téve oktatási és gyakorlati demonstrációkra.

MI-hez intézett kérdések (promptok):
- „Mérési eredmények naplózása.”
- „Letöltési és feltöltési sebesség mérése.”
- „HTTP válaszidő mérési funkció.”
- „Traceroute funkció hozzáadása.”
- „Netstat adatok megjelenítése.”
- „Helyi hálózat pásztázása.”
- „UPnP porttovábbítás kezelése.”

Fejlesztési megközelítés: Az új funkciók fejlesztését a hallgató és a MI tematikusan, modulonként végezte, az egyes feladatokra külön fókuszálva. *Először a hálózati sebességmérés* került terítékre: a MI javaslatot tett arra, hogyan lehet mérni a letöltési és feltöltési sebességet. Az egyik ajánlás az volt, hogy a program töltsön le egy meghatározott méretű állományt egy közeli szerverről (pl. egy ismert tesztfájlt), és mérje ennek időtartamát, majd számítsa ki a sávszélességet. Hasonlóképpen, a feltöltési sebesség mérésére egy adott mennyiségű adat visszaküldését javasolta a MI egy szerverre, illetve alternatívaként a hallgató kipróbált egy külső webes API-t is (pl. egy nyilvános speedtest szolgáltatást) a MI útmutatása alapján. *A HTTP válaszidő méréséhez* a MI a Java beépített HTTP kliens komponenseit ajánlotta: a fejlesztő a

`HttpURLConnection` osztállyal valósította meg, hogy egy adott URL-hez kérést küldjön, és a válasz megérkezésekor időbélyegek alapján kiszámítsa a késleltetést. Ezt először kiszárított környezetben (pl. egy gyors válaszú helyi szerverrel) próbálták ki, majd internetes címekkel is, figyelve az esetleges időtúllépésekre és hibakezelésre.

Ezután következtek a hálózati diagnosztikai eszközök: a **traceroute** funkció megvalósítása komolyabb tervezést igényelt. A MI részletesen ismertette a traceroute működési elvét: egyre növekvő TTL (Time To Live) értékű csomagok küldése, és az út közben válaszoló routerek ICMP üzeneteinek (TTL expired) figyelése. A hallgató a MI által vázolt algoritmust implementálta: egy ciklusban növelte a csomag TTL-jét, minden lépésben elküldött egy UDP csomagot a cél felé egy nem létező porttal (így a cél vagy egy közbeeső router visszaküld egy ICMP üzenetet), és figyelte az érkező választ. Ehhez a MI javasolta a `DatagramSocket` konfigurálását úgy, hogy el lehessen olvasni a routerek válaszait, illetve felhívta a figyelmet, hogy a Windows és Linux rendszereken eltérő lehet az ICMP üzenetek kezelése. A fejlesztő ennek megfelelően implementált egy traceroute folyamatot, ami listába gyűjti az útvonal csomópontjait. A **netstat**-jellegű funkciónál a MI egy egyszerűbb utat javasolt: a program meghívhatja a rendszeren elérhető `netstat` parancsot, majd a kimenetet feldolgozhatja. A hallgató ezt ki is próbálta: a `Runtime.getRuntime().exec()` segítségével lekérte a `netstat` eredményét, majd parsolta a releváns sorokat (pl. aktív TCP kapcsolatok) és megjelenítette egy szövegdobozban vagy táblázatban. Mivel ez platformfüggő lehet (Windows-on és Linux-on más a netstat kimenete), a MI segítséget nyújtott abban, hogyan ismerje fel a program futás közben, melyik OS alatt van, és ennek megfelelően kezelje a formátumot. A **helyi hálózat pásztázása** funkciót a hallgató a MI ötletei alapján úgy valósította meg, hogy a program egy kiválasztott IP tartomány (pl. 192.168.0.1–254) minden címére küld egy ping kérést vagy nagyon rövid UDP üzenetet, és figyeli, honnan kap választ. A MI tanácsolta párhuzamos szálak használatát ennél, hogy a pásztázás gyorsabb legyen – így a hallgató egy szálmedencét hozott létre, és egyszerre több cím ellen küldött probe-ot, majd az eredményeket összegyűjtötte. Végül az **UPnP porttovábbítás** következett: a MI felhívta a figyelmet arra, hogy a nyers UPnP protokoll implementálása bonyolult lehet, de léteznek külső könyvtárak (pl. *Cling* nevű Java UPnP könyvtár), amelyek leegyszerűsítik ezt. A hallgató ezért letöltött egy ilyen library-t, és a MI útmutatása alapján integrálta a projektbe. Ennek segítségével kísérleti jelleggel sikerült elérni, hogy a program képes legyen megkeresni a hálózaton lévő UPnP kompatibilis routert, és kérni tőle egy meghatározott port továbbítását a helyi gépre (például a csomagküldési teszthez, ha kívülről akarnák elérni). Minden egyes új modul implementálása után intenzív tesztelés következett: a hallgató kipróbálta a traceroute-ot ismert célokra (pl. google.com), futtatta a netstat lekérdezést, a pásztázást egy kisebb alhálózaton, stb., és a MI segítségével javította a felmerülő hibákat (pl. ha a traceroute egyes lépései timeoutoltak, vagy ha a netstat kimenet egyes soraival gond volt a feldolgozás során).

Alkalmazott eszközök: Ebben a fázisban a projekt számos új eszközzel és technológiával bővült. A **sebességméréshez** a Java hálózati I/O és időmérés eszközeit használták: a letöltési teszt egy HTTP kapcsolaton keresztül egy ismert méretű fájl lekérésével valósult meg (a letöltés idejét a rendszer órájával mérték), a feltöltés pedig hasonló módon, vagy egy külső API bevonásával. A **HTTP válaszidő** méréséhez a

`java.net.HttpURLConnection` osztályt használták, ami lehetővé tette HTTP kérések küldését és a válasz fejlécek fogadását; a kezdés és befejezés időpontját rögzítve számolták ki a válaszidőt. A **traceroute** implementációhoz mélyebben bele kellett nyúlni a hálózati kommunikációba: a `DatagramSocket` mellett használatba vették annak alacsony szintű beállításait (például a `socket.setSoTimeout()` a várakozási idő korlátozására minden TTL lépésnél). Bizonyos esetekben a program a rendszer natív parancsát is meghívta (pl. Linux-on a `traceroute`, Windows-on a `ping -traceroute` vagy `pathping`), hogy összehasonlítsa az eredményeket a saját implementációval – ehhez a **ProcessBuilder**/`exec()` hívások adtak lehetőséget, majd az eredményt a program elemezte. A **netstat** funkcióhoz hasonlóképpen a rendszer parancsait használták: Windows-on a `netstat -ano` kifuttatása, Linuxon a `netstat -tulpen` (vagy az újabb `ss` parancs) eredménye alapján listázták a kapcsolatokat. A program felismerte az operációs rendszert (ezt a Java `System.getProperty("os.name")` értékéből tette), és ennek megfelelően választotta ki a futtatandó parancsot és a szövegfeldolgozás módját. A **LAN pásztázásnál** a Java `ExecutorService` került bevetésre: egy rögzített méretű thread pool-ban futottak a ping feladatok, amik az `InetAddress.isReachable()` metódust hívták meg sorban különböző IP-ken. Az így párhuzamosított scan viszonylag gyors eredményt adott arról, mely címek aktívak. Az **UPnP** funkcióhoz integrált *Cling* könyvtár (vagy hasonló) segítségével a program magas szintű API-t kapott a routerrel való kommunikációra: néhány hívással fel tudott deríteni egy internet-átjárót és igényelt rajta portnyitást. Ehhez a külső eszközhöz a gradle/maven projektfájlban kellett hozzáadni a dependency-t (amit a MI segített megkeresni és a megfelelő verziót beállítani). Mindezek az eszközök együtt eredményezték, hogy ebben a fázisban a projekt túllépett a pusztán Java standard könyvtár használatán, és **külső rendszerszintű eszközöket**, valamint **harmadik féltől származó könyvtárakat** is integrált a kitűzött célok elérése érdekében.

Nehézségek: A kiterjesztett funkciók implementálása során merültek fel a legösszetettebb technikai kihívások. A sebességmérésnél például figyelembe kellett venni az internetkapcsolat ingadozásait: ugyanazon mérés többszöri lefuttatása eltérő eredményeket adhat, így dönteni kellett, hogy milyen metodikával (átlagolás, több szálon mért párhuzamos forgalom, stb.) prezentálják az eredményt a felhasználónak. Ebben a MI tanácsa az volt, hogy egyszerű első verzióban egy mérést is elég futtatni, majd jelezni, hogy az pillanatnyi érték. A HTTP válaszidő mérésnél kihívást jelentett a **timeout-ok kezelése**: lassú szerver esetén a programnak várakoznia kellett egy ideig, majd feladni a próbálkozást. Ezt a hallgató a MI javaslatára paraméterezhetővé tette (be lehetett állítani egy maximum várakozási időt a válaszra). A traceroute funkció implementációja számos próba-hibával járt: bizonyos hálózatokon az ICMP üzenetek blokkolva vannak, így a program nem kapott választ – a MI segített felismerni, hogy ilyen esetekben is le kell kezelni (pl. `* * *` jelenjen meg a hop helyén, ahogy a klasszikus traceroute programok is teszik). Platformfüggő nehézség volt, hogy Windows alatt az ICMP csomagok kezeléséhez eltérő megoldást kellett alkalmazni (pl. nyers ICMP csomag küldéséhez speciális jogosultság kell, ezért maradt a UDP-alapú megközelítés). A netstat információk lekérdezésénél a programnak meg kellett birkóznia a parancs kimenetének feldolgozásával: a MI segítségével regex mintákat vagy egyszerű szövegfeldolgozó logikát írtak, de így is volt olyan eset, hogy a formátum váratlan volt (például a helyi cím feloldódott hostnévvé). A hallgató ezeket iteratívan

javította. A helyi hálózat pásztázásakor a párhuzamos végrehajtás hangolása igényelt figyelmet: ha túl sok szál indult egyszerre a pingekhez, az a gépet is megterhelte és a hálózati forgalmat is indokolatlanul megnövelhette. A MI javasolta, hogy korlátozzák a párhuzamos szálak számát (például egyszerre maximum 50 ping fusson), ezt be is vezették. Az UPnP integrációnál a megfelelő könyvtár kiválasztása és használata is tartogatott meglepetéseket: a MI által javasolt példakód nem működött elsőre, mert a hálózati környezet (pl. a router) nem támogatta a funkciót vagy szükség volt a hálózati interfész explicit megadására. Ezeket a hallgató folyamatosan egyeztette a MI-val, és végül sikerült működésre bírni a kritikus részeket. Összességében ez a fázis volt a legösszetettebb és időigényesebb, mivel sok különböző területet ölelt fel; a generatív MI itt is kulcsszerepet játszott abban, hogy a hallgató gyorsan tudjon új információkat elsajátítani (pl. egy könyvtár használatát vagy egy algoritmus működését), és a problémákra megoldásokat találjon.

Eredmény: Az alkalmazás funkciókészlete jelentősen kibővült és a Network Monitor egy **komplex hálózati diagnosztikai eszközzé** vált. A felhasználó immár lefuttathat egy traceroute elemzést, amely megmutatja az útvonalat a kívánt célállomáshoz (a köztes routerek listájával és késleltetéseikkel). Lekérdezheti a gép aktuális hálózati kapcsolatainak listáját a beépített netstat funkcióval, ami segíthet megérteni, milyen portok vannak nyitva vagy mely külső szerverekhez kapcsolódik a gép. A program képes pásztázni a helyi hálózatot és felsorolni az aktív IP-címeket, ami hasznos lehet például egy laborban a hálózaton lévő eszközök detektálásához. A beépített sebességmérő megmutatja az internetkapcsolat letöltési/feltöltési sávszélességét, a HTTP teszt pedig a webes válaszidőket mérheti fel, ami webfejlesztési vagy hálózatdiagnosztikai szempontból is érdekes adat. Az UPnP funkció révén (kísérleti jelleggel) a program akár automatizáltan is megpróbálhat portot nyitni a routeren, bár ezt óvatosan kell használni. Ezekkel a kiterjesztett funkciókkal a Network Monitor immár nem csupán egyszerű ping eszköz, hanem egy multifunkcionális program, amely több szempontból is vizsgálni tudja a hálózatot. Az eredmény különösen értékes oktatási környezetben: a diákok egyetlen alkalmazáson belül próbálhatnak ki számos hálózati műveletet és elemezhetik azok eredményét.

VII. fázis – Felhasználói élmény

Célok: A szoftver felhasználói élményének (UX) javítása, hogy a sokrétű funkcionalitás ellenére az alkalmazás könnyen kezelhető és áttekinthető maradjon. Ennek a fázisnak a célkitűzései a következők voltak: valós idejű **grafikonok** beépítése a mérési eredmények szemléletes megjelenítéséhez, a **sötét mód** támogatásának hozzáadása a felülethez (a felhasználói preferenciák és kényelem érdekében), valamint általánosságban a UI (User Interface) elemeinek finomhangolása, hogy az alkalmazás átláthatóbb és esztétikusabb legyen. További célként jelent meg a **mérési adatok exportálása**, hogy a felhasználó elmenthesse az eredményeket későbbi elemzésre vagy dokumentálásra. Összességében e fázis célja az volt, hogy a program ne csak tudásában, de megjelenésében és használhatóságában is megfeleljen egy professzionális eszköz elvárásainak.

MI-hez intézett kérdések (promptok):
- „Valós idejű grafikonok megjelenítése.”
- „Mérési adatok exportálása.”
- „Sötét mód támogatása.”

Fejlesztési megközelítés: A felhasználói élmény javítását célzó fejlesztések párhuzamosan zajlottak, kisebb, egymástól független modulokként a MI tanácsait követve. *A valós idejű grafikonok* bevezetéséhez a MI a JavaFX beépített chart komponenseit (például `LineChart` osztály) javasolta. A hallgató az alkalmazás felületén létrehozott egy új panelt, ahol a grafikon kapott helyet, és beállította, hogy a mérési eredmények (pl. ping idők vagy sávszélesség értékek) folyamatosan frissüljenek rajta a háttérben futó mérések közben. Ehhez meg kellett oldani, hogy minden új mérési adat hozzáadásakor a frissítés a JavaFX fő szálán történjen – a MI emlékeztette a fejlesztőt a `Platform.runLater()` használatára, így a háttérszál a mérési eredményt ezzel a hívással adta át a grafikon komponensnek. Kezdetben a grafikon minden egyes új értéknél bővült, de hamar kiderült, hogy hosszabb végtelenített mérésnél ez túlzsúfoltá teheti a grafikont, ezért a MI tanácsára a fejlesztő úgy módosította, hogy csak az utolsó N adatpont legyen mindig látható (pl. egy csúszó ablakot alkalmazva az adatsorban).

A mérési adatok exportálásának megvalósításakor a MI javaslatot tett a Java fájlkezelő műveleteinek használatára. A hallgató készített egy „Export” gombot, melynek megnyomásakor a program a memóriában tárolt mérési eredményeket (pl. a grafikon adatsorát vagy a ping eredmények listáját) kiírja egy CSV fájlba. A MI segített a megfelelő formátum kialakításában (például első sorba fejléc, utána minden sorba egy időbélyeg és érték páros), illetve abban, hogyan nyissa meg a fájlt írásra (FileWriter segítségével) és kezelje az esetleges IOException kivételeket. A fejlesztő az export funkciót több modulra is kiterjesztette: lehetőség lett például az összes ping eredmény, a traceroute lista, vagy a netstat kimenet exportálására is, mindegyik külön fájlba, így az oktató vagy a felhasználó később analizálhatja az adatokat más eszközökkel.

A sötét mód implementálásához a MI a JavaFX **CSS** stíluslapjainak használatát ajánlotta. A hallgató készített egy külön stílusfájlt, amely a JavaFX alapértelmezett világos témáját felülírja sötét színekkel (háttér, betűk, vezérlők színeinek módosításával). A MI tanácsot adott abban is, hogyan lehet a futásidő alatt váltani a témák között: a fejlesztő egy checkboxot vagy menüopciót hozott létre „Sötét mód” felirattal, és ennek állapotát figyelve a program hozzáadja vagy eltávolítja a megfelelő CSS fájlt a fő jelenet (`Scene`) stíluslistájáról. Az iterációk során a hallgató finomhangolta a sötét téma egyes elemeit (pl. grafikon vonalak színe, táblázatok háttére), és a MI segített pár nehezebben elérhető komponens (mint például az alapértelmezett ablak címsorának) színbeállításaiban is, amennyire a JavaFX engedte.

Alkalmazott eszközök: A grafikonok megjelenítéséhez a JavaFX **Chart API**-ját használták, különösen a vonaldiagram komponenseket (`LineChart`, `XYChart.Series`). A valós idejű frissítés érdekében a már meglévő többszálú architektúrát kombinálták a JavaFX UI-frissítési mechanizmusával. Az adatok tárolására és frissítésére listákat vagy queue-kat alkalmaztak, amelyekből a grafikon mindig az aktuális tartalmat jeleníti meg. Az export funkcióhoz a Java **I/O könyvtár** (java.io) eszközeit vették igénybe: `FileWriter` vagy `BufferedWriter` segítségével a program .csv kiterjesztésű

szövegfájlba írta az adatokat. Ehhez a felhasználónak meg kellett adnia egy fájlnevet vagy elérési utat – ezt egy fájlválasztó dialogus könnyítette meg (a JavaFX `FileChooser` osztályával, amelyet szintén integráltak a MI javaslatára, hogy ne fix fájlba írjon, hanem rákérdezzen a mentés helyére). A sötét módhoz a JavaFX **CSS** mechanizmusa került alkalmazásra: a készített stíluslapot (pl. `dark-theme.css`) a program betöltötte. Ebben a fázisban tehát főleg a felülethez kötődő eszközök domináltak: GUI komponensek, fájlkezelő dialógus, és stíluslapok.

Nehézségek: A valós idejű grafikon implementálásakor figyelni kellett a teljesítményre: ha túl gyakran vagy túl sok adatponttal frissül a grafikon, az a felület akadozásához vezethet. A MI figyelmeztetett erre, és javasolta, hogy limitálják a frissítések gyakoriságát (például másodpercenként legfeljebb 2-5 frissítés). A hallgató ennek megfelelően be is vezetett egy időzítést a háttérszálon, hogy ne próbálja meg túl sűrűn frissíteni a grafikont. Az exportálásnál kihívás volt biztosítani, hogy a fájlformátum mindenki számára olvasható legyen (pl. Excelben megnyitható CSV), és hogy a program kezelje az esetleges hibákat, mint a sikertelen fájlírás. A MI adott mintakódot a try-catch használatára és a hibaüzenetek megjelenítésére (pl. alert ablak a GUI-n, ha nem sikerült menteni). A sötét mód kapcsán a problémát az jelentette, hogy a JavaFX néhány komponensének kinézete nem triviálisan testreszabható (például a natív ablakkeret vagy a görgetősávok bizonyos részei). A MI több ötletet is felvetett, mint például egy harmadik fél által készített sötét téma használatát, de végül a saját CSS mellett döntöttek. A hallgató tesztelte a sötét módot különböző platformokon is, mert előfordulhatnak platform-specifikus eltérések a megjelenésben. Apróbb nehézségként merült fel az is, hogy a sötét/világos mód állapotát meg kellett jegyezni (pl. ha a felhasználó átvált sötét módra, az alkalmazás induláskor is abban induljon-e), de ezt a projekt rövid időtávja miatt végül nem építették be, maradt a futás alatti váltási lehetőség. Összességében a UX fejlesztéseknél a legnagyobb kihívás a **finomhangolás** volt – sok apró részletre kellett figyelni, hogy a végeredmény harmonikus és professzionális legyen.

Eredmény: Az alkalmazás felhasználói élménye jelentősen javult. A valós idejű grafikonoknak köszönhetően a felhasználó azonnali, vizuális visszajelzést kap a hálózati mérésekről: például egy folyamatos ping tesztnél a grafikonon láthatja az idők alakulását, vagy a sebességtesztnél a letöltési sebesség változását. Ez nemcsak látványos, de segíti az adatok értelmezését is. A sötét mód támogatása növeli az alkalmazás komfortosságát: sötét környezetben vagy éjszakai használat során a felület kevésbé fárasztó a szemnek, és választható opcióként professzionálisabb benyomást kelt, hogy a program alkalmazkodik a felhasználói igényekhez. A mérési adatok exportálása lehetővé teszi, hogy a felhasználó megőrizze az eredményeket, vagy akár beadandó riporthoz, hibaelemzéshez felhasználja azokat – ez különösen oktatási környezetben fontos, ahol a diákoknak dokumentálniuk kell a megfigyeléseiket. Emellett a felület általános finomítása (átláthatóbb elrendezés, logikusan csoportosított funkciók, egységes dizájnelemek) azt eredményezte, hogy a Network Monitor mostanra egy **jól áttekinthető, modern megjelenésű** alkalmazássá vált. A hetedik fázis végére tehát a program funkcionalitása mellett a használhatósága és megjelenése is elérte a kitűzött szintet.

VIII. fázis – Dokumentáció és leadás

Célok: A projekt lezárása és a hivatalos követelmények teljesítése. Ebben a fázisban a cél a teljes fejlesztési munka eredményeinek összegyűjtése, rendszerezése és átadása. Konkrétan: a program végleges **forráskódjának csomagolása** (pl. ZIP fájlba) a könnyű továbbítás és értékelés érdekében, egy részletes **PDF dokumentáció** elkészítése, amely bemutatja a projekt célját, működését és fejlesztési folyamatát, valamint külön kiemeli a generatív MI használatát a fejlesztés során. A dokumentáció célja, hogy a tanár vagy bíráló a beadott anyag alapján teljes körű képet kapjon arról, mit valósított meg a hallgató és hogyan. Ehhez kapcsolódó cél volt a kód minőségének végső ellenőrzése is: a program legyen jól strukturált, kommentekkel ellátott, és feleljen meg az oktatási követelményeknek (azaz könnyen magyarázható legyen egy védésen).

MI-hez intézett kérdések (promptok):
- „Forráskód ZIP formátumban történő átadása."

(Megjegyzés: A dokumentáció elkészítéséhez kapcsolódóan nem konkrét programozási prompt, hanem a hallgató saját feladata volt a logok és fejlesztési lépések összegyűjtése; ugyanakkor a generatív MI-t itt is segítségül hívta a tartalom megfogalmazásához.)

Fejlesztési megközelítés: A záró fázisban a hallgató áttekintette a teljes projektet és az összes korábbi fázis anyagát (kódrészletek, MI-val folytatott párbeszédek, jegyzetek). A generatív MI itt is szerepet kapott, de más jelleggel: a kód felülvizsgálatában és a dokumentáció megszerkesztésében segédkezett. Először a hallgató végrehajtott egy **kódtisztítást**: kiszedte a felesleges vagy teszt célú részeket, egységesítette a névkonvenciókat és ahol kellett, ott kommenteket írt a kódba, hogy mások számára is érthető legyen a logika. Ebben a folyamatban időnként kikérte a MI véleményét is, például megkérdezte, hogy a kód bizonyos része érthető-e vagy javasol-e szebb megoldást – a MI több javaslatot is adott (pl. egy bonyolultabb feltétel egyszerűsítésére, vagy egy duplikált kódrészlet kiváltására függvénnyel), és a hallgató ezek közül néhányat implementált, javítva ezzel a kód minőségét. Ezt követően került sor a **dokumentáció összeállítására**. A hallgató összegyűjtötte a teljes fejlesztési napló releváns részleteit: az összes érdemi promptot, amit a generatív MI-nak feltett, és az azok nyomán végzett fejlesztési lépéseket. A dokumentumot fázisokra bontva szerkesztette meg (ahogy ebben a leírásban is látható), és minden fázisnál bemutatta a célokat, a MI-nak feltett kérdéseket, a megvalósítás módját, az alkalmazott technológiákat, a felmerült kihívásokat és a kapott eredményeket. A generatív MI segített a szövegezésben is: a hallgató angol nyelvű források alapján vagy a saját magyar nyelvű vázlatai szerint megfogalmazott bekezdéseket a MI-val ellenőriztette, illetve megkérte, hogy javasoljon professzionális megfogalmazást bizonyos részekre. A kapott javaslatokat a hallgató *kritikusan átnézte és szerkesztette*, hogy a végeredmény egy egységes stílusú, saját hangvételű anyag legyen. A dokumentáció tartalmának véglegesítése során több iteráció történt: a hallgató először csak bullet pointokban összeszedte a fontos tudnivalókat, majd a MI segítségével kibővítette azokat folyó

szöveggé, végül ismét ő maga ellenőrizte, pontosította, hogy szakmailag helytálló és a saját projektjére szabott legyen minden állítás.

Alkalmazott eszközök: A forráskód csomagolásához nem volt szükség különleges eszközre; a hallgató a megszokott ZIP formátumot választotta, amely platformfüggetlenül kibontható. A dokumentáció elkészítéséhez szövegszerkesztő eszközt és a generatív MI nyelvi modelljét használta. A vázlatok összeállítása Microsoft Word-ben vagy Markdown formátumban történt (a hallgató preferenciája szerint), majd a végleges változat PDF-be lett exportálva. A generatív MI (pl. ChatGPT modell) a dokumentáció nyelvi és strukturális ellenőrzésében, valamint a promptok normalizált megfogalmazásában játszott szerepet. A hallgató ezenkívül verziókezelő rendszert is használt a kódhoz a fejlesztés során (Git), ami a leadáskor segített abban, hogy a legfrissebb, működő kód kerüljön a ZIP-be. Az eszköztár tehát ebben a fázisban főként dokumentációs jellegű volt, kiegészülve a MI nyújtotta asszisztenciával a tartalom ellenőrzésében.

Nehézségek: A projekt záró szakaszában a legfőbb kihívás a rendkívül sok információ **összegzése és rendszerezése** volt. A fejlesztés heteiben/hónapjaiban rengeteg apró részlet merült fel, és ezekből ki kellett emelni a lényeget a dokumentáció számára. A hallgatónak végig kellett gondolnia a teljes folyamatot, hogy semmi fontos ne maradjon ki: mely funkciók készültek el, milyen problémák merültek fel és hogyan oldódtak meg, illetve a MI milyen szerepet játszott mindebben. A generatív MI használata a dokumentációban is segített, de egyúttal új kihívást teremtett: a nyers MI által generált szöveg olykor eltért a hallgató saját stílusától vagy netán félreértette a konkrét projekt részleteit, így ezeket minden esetben korrigálni kellett. Fontos volt ügyelni a **konzisztenciára** is: a dokumentumban használt terminológia (pl. MI vs. AI, szál vs. thread stb.) egységes legyen, és a fázisok szerkezete végig következetes maradjon. A hallgató ennek érdekében minden fázist hasonló logikai felépítéssel írt le, ahogy ez a dokumentum elején kitűzött cél is volt. Emellett figyelembe kellett venni a terjedelmi szempontokat: olyan részletes anyagot kellett készíteni, ami a tanári értékelést maximálisan támogatja, de közben áttekinthető marad és nem merül el indokolatlanul a kódrészletekben (a kódot mellékletként, külön fájlban adták át). Végül a hallgató alaposan lektorálta a kész szöveget, hogy ne maradjanak benne nyelvi hibák vagy félrevezető megfogalmazások – ebben szintén segített a MI, de a végső ellenőrzés és felelősség az emberi fejlesztőé volt.

Eredmény: A projekt minden eleme összeállt a leadáshoz. A forráskód strukturáltan, egy ZIP fájlban került átadásra, amely tartalmazza a teljes projektet (forráskód, esetleges erőforrások, build fájlok). A kód megfelel az elvárt minőségi követelményeknek: áttekinthető és jól kommentált, ami megkönnyíti a tanár számára a megértését és értékelését. A mellékelt **PDF dokumentáció** (jelen dokumentum) részletesen bemutatja a Network Monitor projekt fejlesztési fázisait, és külön hangsúlyt fektet a generatív mesterséges intelligencia bevonására a fejlesztés során. Minden fázis külön oldalakon, jól tagoltan jelenik meg, így a bíráló könnyen követheti a projekt fejlődését az ötlettől a megvalósításig. A dokumentum elején világosan szerepel a hallgató neve, Neptun-kódja és a dokumentum célja, ahogy az elvárás volt. Összességében a VIII. fázis eredményeként a hallgató sikeresen lezárta a projektet: leadta a működő, sokoldalú hálózatmonitorozó alkalmazást és a hozzá tartozó átfogó

dokumentációt. Ezzel bizonyítja, hogy nemcsak a technikai megvalósításokban jártas, hanem képes a fejlesztési folyamatot is reflektáltan bemutatni – beleértve azt is, hogyan használta fel a mesterséges intelligencia nyújtotta segítséget a munka során.

Összegzés

A Network Monitor projekt fejlesztése során a hallgató nyolc, jól elkülöníthető fázisban valósította meg az alkalmazás kitűzött funkcióit és javította annak minőségét. A folyamat szerves részét képezte a generatív mesterséges intelligenciával való együttműködés: a hallgató minden fontos fejlesztési lépésnél tudatosan vette igénybe a MI támogatását. A dokumentált promptok és a hozzájuk tartozó fejlesztési eredmények azt mutatják, hogy az MI nem önállóan oldotta meg a feladatokat, hanem **mérnöki asszisztensként** funkcionált. A hallgató adta meg a célokat és követelményeket, a MI pedig javaslatokat, kódmintákat vagy hibaelhárítási ötleteket szolgáltatott, amelyeket a hallgató értékelt és integrált a projektbe.

A fejlesztés kezdetén a MI segített az ötlet formálásában és a megfelelő technológiák kiválasztásában. Később a konkrét funkciók implementálásánál (ping, csomagküldés, különböző -cast módok) gyors forrásként szolgált a Java megvalósításokhoz, felgyorsítva a tanulási görbét. Az olyan összetett feladatoknál, mint a szálkezelés, végtelenített ciklus vagy traceroute algoritmus, a MI lényeges figyelmeztetésekre és bevált gyakorlatokra hívta fel a figyelmet (pl. volatile használata, TTL kezelés), megelőzve ezzel sok potenciális hibát. Ugyanakkor a hallgató döntései és kreativitása végig kulcsszerepet játszottak: az MI által javasolt megoldásokat adaptálni kellett a projekt specifikus kereteihez, és nem egyszer a hallgató próbálkozott több úton is (saját ötlet és MI javaslatának kombinációja) mire megtalálta az optimális megoldást.

A projekt végére egy teljes értékű, sokoldalú hálózatmonitorozó alkalmazás született, amely sikeresen integrálja a generatív AI által nyújtott előnyöket a hagyományos fejlesztési folyamattal. A hallgató ezzel nemcsak a technikai feladatokat oldotta meg, hanem demonstrálta a mesterséges intelligencia értő használatát is egy szoftverfejlesztési projektben. A tanulság az, hogy a megfelelően irányított MI **felgyorsíthatja a fejlesztést**, segíthet a hibák feltárásában és új ötletek generálásában, de a végeredmény minősége továbbra is a fejlesztő szaktudásán, ellenőrzésén és döntésein múlik. A Network Monitor projekt és jelen dokumentáció jól szemlélteti ezt az együttműködést, így támogatva a tanári értékelést és rámutatva arra, hogy a generatív AI hatékony eszköz lehet az oktatásban és fejlesztésben, ha tudatosan és kritikusan használják.
