COMP.SE.110-2025 Software Design

Java Juggernauts

Crypto News app

Design Document

Antti Santala
Veeti Salminen
Onni Rinne
Valtteri Rinne

Contents

1. Project idea

In this chapter we introduce the main idea of the project application and the use cases and requirements.

1.1. App idea

The application is a Java-based cryptocurrency information service that allows users to easily follow the price development of the five largest cryptocurrencies in the market. The app features a dashboard screen that includes three sections: crypto list, crypto detail view with price and volume charts as well as basic information, and a section for major current crypto-related news and adoption topics. The app can be used to follow five top coins, currently: Bitcoin, Ethereum, Tether, XRP and BNB. User is able to select a crypto token, that is displayed in the detail section with historical price and volume data charts. Time frame of the chart can be adjusted. Alongside the chart, the application displays news specifically related to the selected token, allowing users to monitor both market data and relevant external factors in one place. The news section also allows toggling the section to show news about all of the featured cryptos in one place. The goal of the service is to provide a simple and accessible way to stay updated on the most important cryptocurrencies while offering a smooth and interactive user experience.

1.2. Use cases and requirements

The primary use case is to enable users to monitor current and historical cryptocurrency data and stay informed about crypto-related news. Users open the app, browse current crypto news on the side section, select a cryptocurrency from the top 5 list, and then view a price chart and related news. The app focuses on simplicity and fast access to the most relevant market information.

Key Functional Requirements:

- Displaying 5 largest cryptocurrencies by market capitalization
- Show current price for the 5 cryptocurrencies
- Show historical price and volume data up to 365 days for a selected cryptocurrency
- Price history chart with adjustable time range
- Volume chart with adjustable time range
- Show global crypto news
- Show crypto-specific news for the selected cryptocurrency

Non-Functional Requirements:

- Performance: Quick data loading and chart rendering
- Avoid unnecessary API calls
- Usability: Minimalistic UI
- Compatibility: Working on common desktop platforms running Java

2. Technologies

In this chapter we introduce the technologies and the relationships between components used by the application.

2.1. Tools, programming language and framework

The application was developed using Java as the main programming language due to its extensive libraries, cross-platform support, and familiarity within the team. The graphical user interface was implemented using JavaFX, enabling the creation of an interactive layout including dynamic price and volume charts. Development was primarily carried out in Visual Studio Code with Java and JavaFX extensions. GitLab was used for version control, collaboration, and version tracking throughout the project. Team communication and coordination were handled via Microsoft Teams and Telegram.

For cryptocurrency market data, the application integrates the CoinGecko API, which provides current pricing information and historical data for chart rendering. News content is retrieved using the Google SERP API, enabling the application to display both general cryptocurrency news and crypto-specific articles based on the user's selection. JSON responses from both APIs are parsed using Java libraries, and mock data was fully replaced with live API responses during development. Additional handling was implemented to manage API limitations through caching and polling strategies in order to reduce unnecessary network requests and improve performance.

2.2. Components

The application will be structured according to the Model-View-Controller (MVC) architecture, which separates UI presentation, business logic, and data handling. This improves many software quality metrics, such as maintainability, testability, and scalability. Our application will have a separate service layer in addition to the MVC architecture. MVC architecture is represented in the component table below as each category is divided into their own group.

Planned and executed components for the application:

Component	**Purpose**	**Interactions**
Application layer		
App	Main entry point for the JavaFX application. It is responsible for instantiating all the major components (Models, Views, Controllers, Services) and wiring them together to form the complete, functional application	Initializes CryptoService, PricePollingService, NewsService, and all controllers/views. Wires event listeners and callbacks between components.
Service layer		
CryptoService	Fetches, caches, and preloads crypto data from CoinGecko API. Parses JSON responses into domain objects. Implements retry logic and double-checked locking for thread-safe caching.	Called by CryptoListController and CryptoDetailController. Returns List<Crypto> and HistoricalData. Uses CryptoCache internally for result caching.
PricePollingService	Polls the lightweight /simple/price CoinGecko endpoint at regular intervals to fetch live price updates for top cryptocurrencies.	Called by App to start/stop polling. Invokes a callback when prices are updated. Updates are reflected in the UI via MainView.
NewsService	Fetches crypto-related news articles from SerpAPI. Supports filtering by specific cryptocurrency or returning	Called by NewsController with optional crypto ID parameter. Returns List<News>. Uses CryptoService to get the latest top cryptos for filtering.

	general crypto news. Parses JSON responses into News model objects	
CryptoCache	Fetches, caches, and preloads crypto data from CoinGecko API. Parses JSON responses into domain objects. Implements retry logic and double-checked locking for thread-safe caching.	Called by CryptoListController and CryptoDetailController. Returns List<Crypto> and HistoricalData. Uses CryptoCache internally for result caching.
ApiConfig	Centralized configuration reader for API keys and endpoint URLs. Loads properties from application.properties.	Used by CryptoService, PricePollingService, and NewsService to access configuration values.
Model layer		
Crypto	Represents a single cryptocurrency. Holds ID, name, symbol, current price, 24h change percentage, market cap, trading volume, and circulating supply.	Passed from CryptoService to CryptoListController, then to CryptoListView for display. Selected crypto is passed to CryptoDetailController and NewsController.
HistoricalData	Container object holding a list of ChartPoint objects representing price/volume history for a cryptocurrency over a specific time period (1D, 7D, 30D, 90D, 1Y).	Returned by CryptoService to CryptoDetailController, then passed to CryptoDetailView for chart rendering.
ChartPoint	Represents a single data point on a time-series chart. Contains timestamp (Instant), price (Double), and volume (Double).	Held within HistoricalData. Used by CryptoDetailView to plot chart data points.
News	Represents a single news article. Holds title, source, link/URL, summary, and publication date.	Returned by NewsService to NewsController, then passed to NewsView for display.
Controller layer		
MainController	High-level orchestrator of application flow. Manages the state of the selected cryptocurrency. Coordinates interactions between specialized controllers. Responds to user actions (selection, toggle events) and delegates tasks.	Holds references to CryptoListController, CryptoDetailController, and NewsController. Listens to events from CryptoListView and NewsView. Notifies detail and news controllers when state changes.
CryptoListController	Manages logic for the cryptocurrency list view. Loads and displays the top 5 cryptocurrencies. Responds to user selection and search interactions.	Calls CryptoService.getTopCryptos() to fetch data. Updates CryptoListView with results. Notifies MainController when a cryptocurrency is selected.
CryptoDetailController	Manages logic for the detailed cryptocurrency view. Fetches and provides historical chart data based on time interval selection (1D, 7D, 30D, 90D, 1Y) and chart type (price vs. volume).	Receives commands from MainController. Calls CryptoService.getHistoricalDataForCrypto() to fetch time-series data. Updates CryptoDetailView with historical data and metadata

NewsController	Manages logic for the news view. Fetches news articles based on application state (general crypto news or news specific to selected cryptocurrency).	Receives commands from MainController. Calls NewsService to fetch news data. Updates NewsView with parsed news articles.
View layer		
MainView	Root UI container for the entire application. Lays out the UI in three columns: crypto list (left), detailed view (center), and news feed (right).	Holds references to and displays CryptoListView, CryptoDetailView, and NewsView. Serves as the primary JavaFX scene.
CryptoListView	Left sidebar UI displaying a searchable, scrollable list of top 5 cryptocurrencies. Shows price, 24h change, and market cap for each.	Receives List<Crypto> from CryptoListController and renders items. Fires selection events to CryptoListController when user clicks a crypto item.
CryptoDetailView	Central panel UI displaying detailed information and historical charts for the selected cryptocurrency. Includes price/market stats and price/volume toggle. Offers time interval selection buttons (1D, 7D, 30D, 90D, 1Y).	Receives HistoricalData and Crypto metadata from CryptoDetailController. Renders line/area charts. Fires events to CryptoDetailController when user changes time interval or toggles price/volume.
NewsView	Right sidebar UI displaying a list of news articles. Includes toggle/filter to switch between general crypto news and news for the selected cryptocurrency.	Receives List<News> from NewsController and renders articles (title, source, date, summary). Fires toggle/filter events to NewsController.

Component diagram focusing on public interfaces and data flow through the application can be found in Appendix A.

2.3. Component methods

This table describes the functional components of the Crypto Dashboard application. UI-only components and passive data objects are summarized, while services, controllers, caching mechanisms, and background processing are described in detail.

Class / Component	**Method / Component**	**Internal Structure & Implementation Details**	**Responsibility / What it does**
App *(Application Entry Point)*	main(String[] args)	Wires components: Services: CryptoService, PricePollingService, NewsService Cache: CryptoCache	Bootstraps the application, wires dependencies, initializes background tasks, and launches the UI (JavaFX lifecycle).

Class / Component	Method / Component	Internal Structure & Implementation Details	Responsibility / What it does
		Controllers: MainController, CryptoListController, CryptoDetailController, NewsController Views: MainView and sub-views. Registers callbacks: Price updates, data load completion.	
config.ApiConfig *(Configuration Loader)*	loadProperties()	Loads /application.properties from the classpath into a static Properties object.	Centralized configuration loading.
	Getters (getCoinGeckoApiKey, getSerpApiKey, etc.)	N/A	Return configuration values with defaults.
services.CryptoCache *(In-memory Cache Layer)*	setTopCryptos, getTopCryptos, hasTopCryptos	Uses synchronized access for the crypto list.	Provides thread-safe caching for top cryptocurrencies.
	putHistoricalData, getHistoricalData, hasHistoricalData	Uses ConcurrentHashMap for historical data.	Provides thread-safe caching for historical market data.
	clear(), getHistoricalDataCount()	N/A	Clears cache or returns diagnostic count.
services.ICryptoService *(Service Interface)*	Interface Definition	Methods: getTopCryptos, hasHistoricalData, getHistoricalDataForCrypto, preloadAllData, setDataLoadedCallback, getFailedLoadsCount, clearCache.	Defines the contract for crypto data loading and caching. Acts as an abstraction layer.
services.CryptoService	Constructor	Multiple constructors allow dependency injection.	Configuration is loaded per instance.

Class / Component	Method / Component	Internal Structure & Implementation Details	Responsibility / What it does
(Core Business Logic)			
	getTopCryptos()	Uses synchronized double-checking.	Returns cached results if available; otherwise triggers fetch.
	fetchTopCryptosFromAPI()	Handles API request creation, API key handling, and retry logic with backoff.	Fetches the top cryptocurrency list from the API.
	parseCoinsJson(String json)	N/A	Parses CoinGecko JSON into Crypto domain objects.
	getHistoricalDataForCrypto(String id, String days)	Validates input, checks cache.	Fetches and stores data if missing from cache.
	fetchHistoricalDataFromAPI(...)	Implements retry logic and backoff.	Fetches market chart data.
	fetchHistoricalDataFromAPINoRetry(...)	Single-shot fetch.	Used specifically for parallel preloading to avoid blocking threads.
	parseMarketChartJson(String json)	Transforms CoinGecko chart JSON into ChartPoint objects.	Returns HistoricalData.
	preloadAllData()	Parallel bulk loader: 1. Fetches top cryptocurrencies 2. Creates fetch tasks for multiple time intervals 3. Executes parallel requests 4. Stores successful fetches into cache 5. Retries failures in batches 6. Notifies UI using callback mechanism.	Orchestrates the massive parallel loading of data.

Class / Component	Method / Component	Internal Structure & Implementation Details	Responsibility / What it does
	retryInBatches(...)	Uses batch control, sleep delays, and capped retry attempts.	Retries failed preloads systematically.
	getFailedLoadsCount()	N/A	Returns total permanently failed loads.
	clearCache()	N/A	Resets cache and diagnostics.
	setDataLoadedCallback(...)	N/A	Registers callback to notify UI of load progress.
	Formatting Helpers	formatMoneyShort(double), formatNumberShort(double)	Formats numbers for display.
services.PricePollingService *(Live Price Updates)*	Construction	Initializes HTTP client, Jackson parser, scheduler, and property loader.	Sets up the environment for polling.
	startPolling(List<Crypto>)	Schedules background polling with a fixed delay.	Starts the periodic update process.
	pollPrices()	1. Queries CoinGecko price endpoint 2. Updates in-memory maps 3. Notifies UI via callback.	Performs the actual fetch and update cycle.
	stopPolling()	Stops executor gracefully.	Ends the background task.
	parsePriceResponse(String json)	N/A	Extracts prices and 24-hour changes from JSON.
	Accessors	getCurrentPrice, getCurrentChange, getAllCurrentPrices, getAllCurrentChanges	Provide access to the latest polled data.
services. NewsService *(News Retrieval)*	getNews(String cryptoId)	N/A	Fetches news for a specific cryptocurrency.

Class / Component	Method / Component	Internal Structure & Implementation Details	Responsibility / What it does
	getGeneralNews()	N/A	Fetches general crypto-related news.
	parseNewsResponse(String json)	N/A	Parses article data into domain News objects.
Controllers *(Application Logic)*	MainController	Coordinates Views, Controllers, Service callbacks, Crypto selection, and App state.	Handles application shutdown and central coordination.
	CryptoListController	Handles top crypto loading, user selection, and search/filter logic.	Manages the list view logic.
	CryptoDetailController	Handles chart updates, interval changes, coin metadata refresh, and price/volume switching.	Manages the detailed view logic.
	NewsController	Routes general or coin-specific news, handles filter toggles, and updates views.	Manages the news view logic.
Models *(Passive Data Structures)*	Crypto, HistoricalData, ChartPoint, News	Contain no business logic. Expose data via constructors and getters only.	Crypto: Metadata. HistoricalData: Chart container. ChartPoint: Timestamp + value tuple. News: Article details.
Views *(UI Layer)*	MainView, CryptoListView, CryptoDetailView, NewsView	Contain no business logic. Act as passive renderers.	MainView: Layout shell. CryptoListView: List + filtering. CryptoDetailView: Chart + metadata. NewsView: Article list + filters. Displays data and forwards user events to controllers.

2.4 Unit testing

We created unit tests for the most critical components of our application. We selected one model to be unit tested because it was the only model with actual logic while the other models are mostly dumb data models. Then the most obvious selections were the services fetching the data from the two APIs. These are very critical to test to ensure the application works as expected as the value of the application is heavily based on external data. The fourth component we selected was the caching service as the data fetching services rely on it and we are working with strict rate limits on the CoinGecko API. Therefore, it is crucial for our caching to work.

The used testing frameworks are JUnit and Mockito. Mockito is used to simulate API responses from our two APIs. No real API calls are made, so we do not depend on external services, and the tests can be run very fast without actual requests. The tested files are presented in the following table.

Test file	Component / Class being tested	What is being tested
CryptoTest.java	Crypto	**Cryptocurrency data model validation**: Tests object construction with all properties (id, name, symbol, price, change percentage, market cap, volume, circulating supply). Validates price and change percentage formatting with appropriate symbols (dollar signs, arrows, percent signs). Tests equality based on ID field, inequality with different IDs and null comparisons. Tests hashCode consistency for objects with same ID.
CryptoCacheTest.java	CryptoCache	**In-memory caching system**: Tests storage and retrieval of top cryptocurrency lists and historical price data for different cryptocurrencies and time intervals. Tests cache state checking methods, clearing operations, and counting cached entries. Verifies defensive copying to prevent external modifications and null-safe operations.
CryptoServiceTest.java	CryptoService	**CoinGecko API integration**: Tests fetching and parsing cryptocurrency data from external API including prices, market caps, volumes, and supply information. Validates caching behavior to prevent redundant API calls. Tests error

		handling for API failures and network errors. Verifies input validation for null/empty parameters. Tests cache clearing and configuration of retry/delay settings. Tests constructor validation for required dependencies.
SerpAPINewsServiceTest.java	NewsService	**Cryptocurrency news aggregation**: Tests fetching and parsing news articles from SerpAPI including titles, sources, links, summaries, and timestamps. Validates filtering of invalid entries with missing required fields. Tests handling of empty results and API error responses. Verifies fallback behavior for missing date information. Tests both cryptocurrency-specific and general news queries. Tests constructor validation for required dependencies.

3. Self-evaluation for midterm submission

In this chapter, we will evaluate how well our design has supported the implementation and how well we are anticipating it to support the implementation of remaining functionality.

3.1 Original plan and design

We have mostly been able to stick to the original design for the most part. Our project structure is now MVC with the additional service layer, as was initially planned. However, the project structure did not fully correspond to the MVC structure at one point and had to be refactored for the midterm submission to follow MVC more strictly. We had to create the controller layer, which separates business logic from the views and works as a bridge between the services and views, passing data models. The views were refactored to be independent and to only be responsible of rendering the UI. This was done as the MVC structure allows the project to have clear separation of concerns, leading to better testability, component reusability, maintainability and scalability. Another change in design was to refactor the CryptoListView into its own view and controller as it was formerly created in MainView. This was likewise done to obtain a clear separation of concerns and good modularity.

We also added a second graph of crypto volumes, in addition to the price graph we already had, to meet the requirements of the group assignment. This volume bar graph was not initially part of our design plans but since we learned we needed another visual representation of data, we quickly decided that a volume bar graph would suit the application and complement the crypto detail data well.

Implementing features has been quite straightforward based on our original plan. We are able to display details about the five largest crypto coins using CoinGecko API and get news about them using the SERP

API, as was initially planned. However, there have also been some minor changes to the plan and requirements. We found out that CoinGecko has rate limits so it seems like the requirement of showing live price could be difficult. Similarly, quick crypto data loading and chart rendering are not currently possible with the rate limits as changing the intervals on the crypto chart quickly consumes API calls, leading to the chart not updating once the rate limit has been reached. We could try caching the data locally to reduce the number of API calls and try polling the live data every X seconds/minutes to overcome these challenges.

Original app idea and requirements were updated to match the current implementation choices and plan for the rest of the implementation. The description about the main view and its sections was updated and cleared up. Requirement about chart zoom and plan on websocket usage were removed due to API limitations and design decisions. We will be exploring polling for the final submission.

3.2. Current design and quality

As previously stated, our current design that follows the MVC architecture has many benefits that improve the quality of the software, including:

- Maintainability: clear structure makes it easy to locate and fix bugs
- Testability: modular design makes testing separate components easier
- Reusability: data model like Crypto can be used by multiple views and controllers
- Scalability: it is easy to add new features as the existing structure can be utilized
- Reliability: changes or errors in one layer is less likely to break others
- Modifiability/flexibility: technologies, APIs or UI can often be changed without changing other logic

3.3. Anticipated changes

We are anticipating some changes to the design and plan as we start implementing the remaining features even though the architecture is already solid and we have most features implemented. We will be creating unit tests for the most critical functionality like data parsing of the APIs and data formatting of the model classes. We likely need to refactor the services to be more testable as currently they use hard coded and static dependencies, making it difficult to mock HTTP responses and test without real API calls. The refactor will likely be done by extracting the interfaces and implementing dependency injection. This way we can mock services and HTTP clients easily and test without actual API calls.

Apart from the tests, we will be refining the existing logic and structure of the application. We will likely implement crypto data caching to reduce the CoinGecko API calls and to make chart rendering smoother. Caching will also allow us to potentially do polling of price / volume data to update charts because of reduced API calls.

4. Self-evaluation for project submission

In this chapter, we evaluate how well the original design supported the full implementation of the application and how the completed system reflects our architectural and technical choices.

4.1 Original plan and design

Overall, the original design has supported implementation very well. The decision to use the MVC architecture with an additional service layer proved effective and helped guide the project development

from early implementation to the final submission. After refactoring the structure at midterm to properly follow the MVC pattern, the overall structure remained stable for the rest of the project and did not require major changes.

The separation of responsibilities between models, views, controllers, and services significantly improved clarity during implementation. Business logic and API communication were cleanly isolated into the service layer, while controllers focused on orchestration and views remained strictly responsible for UI rendering. This separation reduced coupling and made debugging and extending the application easier.

The component structure defined earlier was successfully implemented. All major components described in the design document were realized, including the CryptoService, PricePollingService, NewsService, controllers, and domain models. Dedicated API services were created to replace earlier mock data, and the UI was connected to live API responses from CoinGecko and SerpAPI. Manual adjustments were made to API parameters and parsing logic to ensure stable and accurate results.

Some minor design changes were required during implementation. Due to API rate limits, the initial idea of continuous real-time updates was revised. Instead, lightweight polling and caching strategies were introduced to reduce the number of API calls and improve user experience. The planned WebSocket approach was fully removed as it was not supported by the selected APIs. However, these changes did not affect the overall architecture and were implemented entirely within the service layer, demonstrating the flexibility of the original design.

4.2. Current design and quality

The final system satisfies the main functional and non-functional requirements set at the beginning of the project. The application successfully displays the top five cryptocurrencies with current prices, historical price and volume charts, and related news content. The user experience is clean, responsive and intuitive.

The MVC structure has brought several quality benefits:

- Maintainability: Clear responsibility boundaries make it easy to locate bugs and introduce fixes.
- Testability: Business logic is isolated from the UI and can be tested independently.
- Reusability: Domain models such as Crypto and News are used across multiple controllers and services.
- Scalability: New features such as additional cryptocurrencies or data sources could be added with minimal restructuring.
- Reliability: Failures in one layer are less likely to affect the whole application.
- Flexibility: APIs, UI components, or implementation technologies can be changed without modifying unrelated modules.

The service layer proved especially useful. API communication, polling, caching and error handling are all centralized, keeping controllers clean and views simple. This made iteration and debugging faster as API-related changes were confined to a single layer.

4.3. Reflection and future improvements

Despite the successful final result, there are areas for improvement. The biggest remaining technical challenge is API dependency. Rate limits restrict data freshness and chart responsiveness. While caching and polling partially solve this, a production-grade version would benefit from paid API subscriptions or alternative data providers.

Another major improvement area is testing. Due to time constraints, comprehensive unit testing was not fully implemented. The services currently depend on hard-coded HTTP clients and static calls, which makes mocking and automated testing more difficult. Future development should introduce interface-based design and dependency injection so services can be tested without real API calls.

Additional future improvements include:

- Adding persistent caching to store API responses locally.
- Introducing loading indicators and better error handling in the UI.
- Supporting more cryptocurrencies or user customization.
- Improving news filtering and sorting mechanisms.
- Adding application packaging and configuration profiles for easier deployment.

Overall, the project demonstrates that the original design was both realistic and robust. The architecture allowed the application to evolve without major rewrites, and design decisions taken early proved to be technically sound. The final result reflects both the learning outcomes of the course and a strong understanding of architectural design practices.

5. Group member contributions

	Antti	Veeti	Onni	Valtteri
Prototype	Created design document and planned the project idea.	Finalizing document. Tweaking the app architecture plan.	Initial UI layout code and components code	UI prototype modifications and finalization (crypto list, news toggle), created README
Midterm	Created API services for CoinGecko and SerpAPI and integrated them into the data handling and UI to display live data. I removed the mock data and added a dedicated	Implemented the historical data part of the crypto service and connected data to the chart. Implemented volume chart. Removed unused parts of the app and updating readme for midterm	Refactor to separate crypto list logic and UI from main view and main controller by adding a separate view and controller for the list. Update document for midterm (updated	Refactor project structure into MVC with services (created controller layer). Update document for midterm (self-evaluation section, minor updates across the document)

	handler file for the APIs	submission. Updating app idea and requirements.	component table and graph).	
Final submission	Added Coingecko API call rate limit fix and fixed news service not properly loading. Added feature where app disables/enables buttons based on if API has provided data. Tweaked README to represent app current status. Updated design document to match final submission status for example: updated components table and created methods table added chapter 4 and updated AI chapter.	Fixed issues with volume chart rendering and overall refined the chart component. Implemented the call retry mechanism for the top cryptos call. Removed hard coded crypto values from newsService. Updated the component diagram of the app.	Implemented price polling to update crypto prices periodically. Implemented unit tests for the most crucial components. Changed crypto API calls to be parallel for fast data fetching. Migrated back to crypto API with key for larger amount of API calls per min. Created unit test section 2.4 to the document.	Refactored services to be easily unit testable (extract interfaces, dependency injection). Made news fetching async to avoid blocking the UI. Made Bitcoin selected as default and added a news skeleton while news is being fetched to improve UX. Improved news search results. Updated design document (section 6.4)

6. Use of AI

6.1. Planning phase AI usage

Artificial intelligence was used during the planning phase of the project to support documentation quality and clarify component relationships. The idea, API selection and technological choices were created independently by the project team, but AI was utilized to improve the structure and clarity of the design documentation. AI was also used in the coding phase.

6.1.1. Code (UI prototype)

The prototype UI code was generated with the assistance of Copilot AI tool. The AI tool was used to produce initial UI layout code and UI components based on the selected app idea and our initial design plans. All generated code was reviewed, and components including the chart and news section were updated by modifying colors, fonts and sizes. New UI components and logic were also introduced including adding the left sidebar and a news filter for selected cryptos. These changes were all designed by us but generated with the AI tool iteratively.

6.1.2. Project planning and Desing document

The design document content was partially produced with the assistance of generative AI tools. The AI was used during the planning phase to support structuring the project concept and improving documentation quality. The project idea, the selected APIs, and the overall technology stack were created independently by the team, and AI was not used to make any decisions regarding features, architecture, or implementation.

AI assistance was used in mapping the relationships between application components based on our own descriptions of the system idea. The tool suggested an initial component breakdown and data flow structure within the MVC architecture. All final design decisions were made by the team, and components were further refined and renamed where necessary to better match the actual implementation plan.

AI was also used in drafting and improving the wording of functional and non-functional requirements based on our own notes. The original text for these sections was written in Finnish and translated into English using the AI tool. Additionally, some paragraphs were rewritten for better clarity and readability, while preserving our intended meaning and content. All documentation produced with AI support was reviewed and updated by the team before inclusion in this document.

6.2. Midterm AI usage

Artificial intelligence was used during the midterm phase of the project to support both development and documentation tasks. AI tools were primarily applied to assist in code refactoring, API data handling and documentation refinement. All AI-generated outputs were reviewed and adapted by the team before being included in the final implementation.

6.2.1. Code

All of the refactoring (creating controller layer and separating crypto list from main view) mentioned in the self-evaluation chapter was generated with Copilot AI tool, in accordance with our own modification plans and goals. The code was then manually reviewed and modified where it was necessary.

In the API integration phase, the API calls for CoinGecko and SerpAPI were implemented manually, but Copilot AI was used to improve the handling of the returned data. AI helped generate the initial logic for parsing, formatting, and replacing the existing mock data with live API responses. A dedicated API handler file was created and AI helped rearrange the code. All API parameters, request structures and data mappings were manually reviewed and corrected to ensure compatibility with the final UI implementation. Connecting the fetched data to the existing mocked graph was mainly implemented with Copilot with giving it granular tasks to get to the end goal of 2 graph options to choose from. Copilot was also used in debugging issues with the chart implementation, but also required manual intervention.

6.2.2. Design document

In the component table, all the purposes and interactions were first generated with AI and then the responses were modified a little by changing wording, excluding certain points and adding some things the AI did not mention. AI was also used to get the data flow arrow directions in the component interactions diagram.

AI was slightly used in the self-evaluation section. It was used to explore solutions for reducing the GoinGecko API calls, which are mentioned at the end of original plan and design section. AI was also

utilized to explore some of the quality benefits of the MVC structure, seen in the current design and quality section. Finally, AI was used to find out how the services can be unit tested easier.

6.3. Final phase AI usage

Artificial intelligence was used during the final phase of the project to assist with documentation refinement and final code improvements. AI functioned as a support tool rather than a decision-maker.

6.3.1. Design document

In the final documentation phase, AI was used to clarify, restructure and improve the clarity of selected sections of the design document. In particular, AI was used to refine the wording and structure of Chapters 4 and 6, ensuring consistent academic tone, improved readability and clearer explanation of technical decisions. AI was also used to generate the content of the table presenting the unit tested components in section 2.4. All content revisions were carefully reviewed and approved by the project team before submission.

All design document related AI usage throughout the assignment was done using ChatGTP or Gemini 3 Pro. The approach used was simply giving the AI the context of the assignment instructions and asking it to create an initial draft of the section / table under work or ask for translation or improvement of wording of a certain section.

6.3.2. Code

In the final coding phase, AI was used for code improvements, targeted refactoring and debugging assistance. This included refining method structures in the service layer, improving data handling logic, and assisting with error handling for API failures and malformed responses. AI was also used to support the caching-related adjustments to reduce unnecessary CoinGecko API calls and improve performance.

Claude Opus 4.5 was used through GitHub Copilot in VSCode to refactor the services to be easily unit testable. The AI model successfully extracted the interfaces and implemented dependency injection. In addition, small UX improvements like making news fetching asynchronous, selecting Bitcoin as default on app start up and adding a placeholder news skeleton while news is being fetched were implemented using the same tool.

The implementation of polling the CoinGecko API for automatically updating the crypto prices, in addition to the unit test creation was entirely implemented with AI tools. The AI also helped to identify the Crypto model and CryptoCache as critical components, while we ourselves picked the data fetching services to be tested. AI was also fully used for code generation to migrate from keyless CoinGecko API to the key version and to make the crypto related API calls parallel.

Additionally, AI was used to verify code consistency after replacing mock data with live API responses from CoinGecko and SerpAPI. This involved validating that domain models correctly matched API responses, ensuring that controllers and views were properly synchronized with updated service logic, and checking that UI components correctly reflected updated data states.

For debugging and improvement of the chart section of the app, Google Antigravity IDE with its agentic AI assistant was used. With Gemini 3 Pro (High) as the model, the agent was very successful in completing tasks that were given to it and most of the issues with results were because of low accuracy prompts. Same tool was used to implement the retry mechanism to the top cryptos api call. The

mechanism was already implemented to the historical data call, so the model was able to successfully transform the same functionality to the top cryptos call. The agent provided implementation plans and reports of the performed tasks, that were verified for correctness along with the code changes.

All AI-generated code was manually reviewed and adapted to ensure correctness, stability and alignment with the existing MVC-based architecture. All final development decisions were made by the project team.

All code related AI usage throughout the assignment was done using Antigravity, Cursor or VS Code as the IDE, and Claude Opus 4.5, Gemini 2.5 Pro or Gemini 3 Pro as the agent. The approach was to proceed in simple steps like one feature at a time and giving the agent instructions based on our design plan.

6.4 Advantages & disadvantages of AI

In the context of this assignment, the utilized AI tools were very beneficial and useful in our opinion. Overall, they made planning and implementing the application efficient and they hardly made any mistakes. Obviously, we reviewed the outputs carefully and refined them when necessary.

There are many advantages of utilizing AI tools in software design in general. The modern AI models have a good understanding of basic design principles like the MVC structure and SOLID principles. They are also great at refactoring code and making structural changes. Thus, they can be utilized to design and implement software more efficiently, at least at a medium scale.

While AI has many benefits in software design, it also has its disadvantages. It is widely known that even state-of-the-art AI models can sometimes hallucinate content and in the field of software design that can be dangerous. If an application is entirely designed by AI tools and the plan contains hallucinated content, it can be very time consuming and expensive to fix those issues once the application is far into development. Secondly, AI models have limited context windows, so making large structural changes in a big codebase could be difficult for AI models and lead to code quality issues. Because of these disadvantages, it is essential that AI tools are used as supportive tools rather than just blindly abusing them.