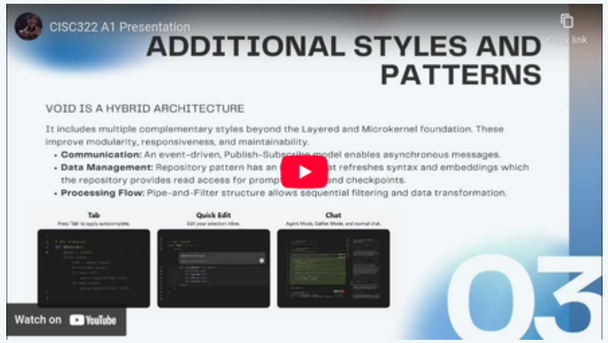


https://youtu.be/4iqmJzOcO6g

VOID IDE

CONCEPTUAL ARCHITECTURE

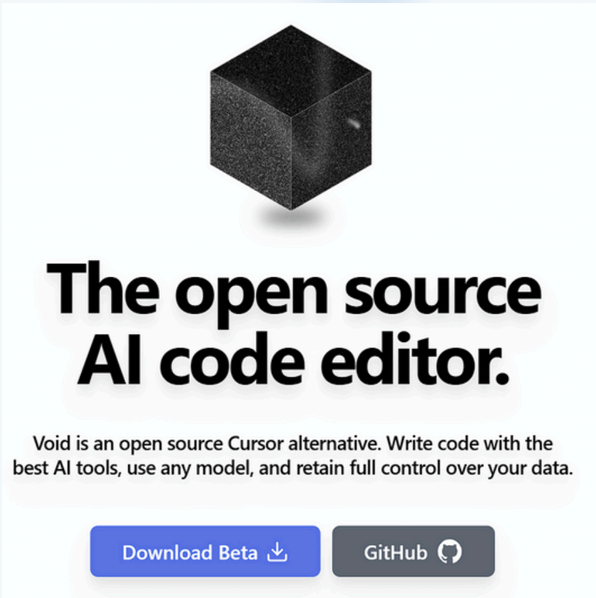
Presentation By: Emily Cheng, Jeff Hong

GROUP MEMBERS

- Jinpeng Deng 22SSl17@queensu.ca – (Group Leader) Conclusions, Lessons Learnt, AI Collaboration.
- Jeff Hong 22RQ20@queensu.ca – (Presenter) Use Cases, Grammar Review, Introduction, Abstract, slides.
- Emily Cheng 22TWS1@queensu.ca – (Presenter) Architecture Styles, Software Layers and Components, Interaction, Major Subsystems, Editing, slides.
- Zijie Gan 21ZG6@queensu.ca – System Evolution, Data Flow.
- Nathan Daneliak 22TMX2@queensu.ca – Functionality, External Interfaces.
- Zipeng Chen 23QH10@queensu.ca – Concurrency, Developer Responsibilities.

01

VOID IDE Report



ARCHITECTURE BREAKDOWN

LLM Messaging Service
Manages sending and returning requests from the user and responses from the AI.

Provider API
Connection to the user chosen API to make requests to.

DiffZones
Component responsible for creating and mapping the different zones where the AI is allowed to interact with the code.

Slow Apply
A component that lets the AI modify entire files of code if requested.

Chat UI
The visual interface that lets users send requests, and displays the responses from the AI.

Main Process
Component that acts like the gate way between the AI and the IDE.

Fast Apply
Lets AI make suggestions about replacements and changes within multiple blocks of code.

Tools System
Gives AI access to control workspace with the user's permission.

02

ARCHITECTURE STYLES

OVERALL...

Void's conceptual architecture combines both **Layered** and **Microkernel** styles. It builds on a multi-layered foundation with Electron as the base desktop environment, VS Code as the extension framework, and Void adding an orchestration layer for AI assistance. This structure allows independent evolution of each layer while maintaining stability and responsiveness.

01 **Hybrid Structure**
Combines Microkernel (Plug-In) as the main structure with a Layered foundation.

02 **Communication and Data Flow**
Event-Driven/Publish-Subscribe Messaging and Pipes and Filter for Processing.

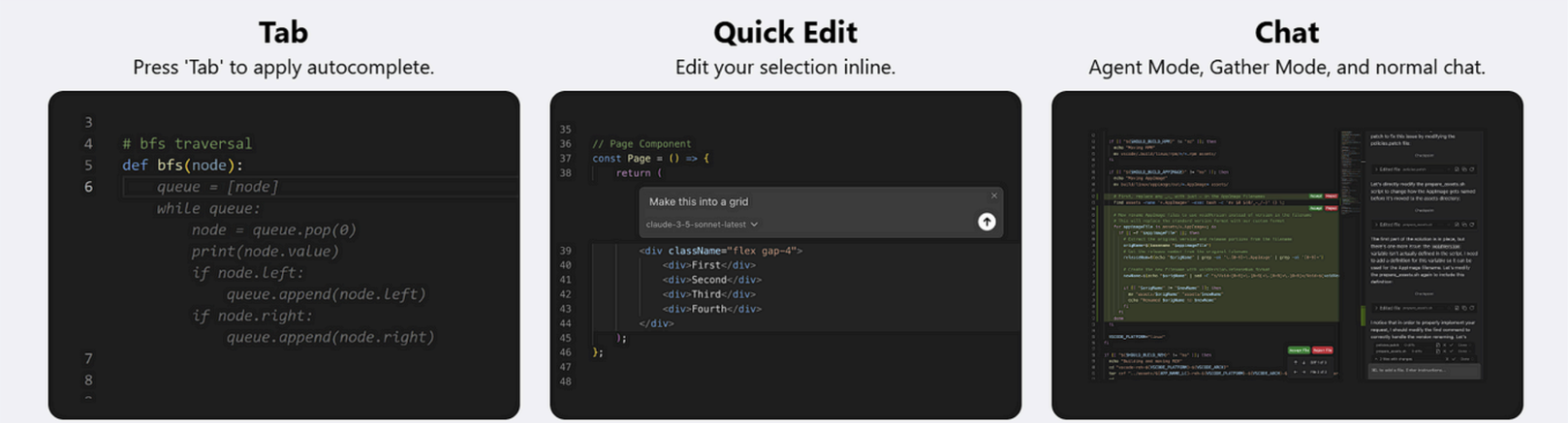
03 **Data Management**
Repository pattern means the indexer refreshes data and repository provides prompt-building and checkpoint access.

ADDITIONAL STYLES AND PATTERNS

VOID IS A HYBRID ARCHITECTURE

It includes multiple complementary styles beyond the Layered and Microkernel foundation. These improve modularity, responsiveness, and maintainability.

- **Communication:** An event-driven, Publish-Subscribe model enables asynchronous messages.
- **Data Management:** Repository pattern has an indexer that refreshes syntax and embeddings which the repository provides read access for prompt building and checkpoints.
- **Processing Flow:** Pipe-and-Filter structure allows sequential filtering and data transformation.



ALTERNATIVE: MONOLITHIC DESIGN

A SIMPLER BUT LESS FLEXIBLE DESIGN THAT TRADES MODULARITY FOR SPEED

Current Design Tradeoffs

- Microkernel design adds Inter-Process Communication (IPC) overhead.
- Debugging is more complex across isolated processes.
- A purely layered or service-oriented style would simplify communication but make parts more tightly coupled, thus reducing modular control and extensibility.

Monolithic Alternative

All functionality embedded directly in VS Code's renderer to reduce IPC overhead.

- Components run in a single process using direct function calls. Simplifies development, deployment, and centralized debugging.
- Tightly couples all modules (harder to scale/integrate new tech).
- Minor changes could require rebuilding the entire system.
- Suitable mainly for small and fast prototypes. Not so much for Void's long-term goals of extensibility, portability, and security.

TOP LEVEL SUBSYSTEMS

FUNCTIONALITY AND RESPONSIBILITIES

01

voidSettingsService
Manages provider configurations and model settings. Acts as an internal dependency supporting all other core services.

02

LLM Messaging Pipeline
Handles communication with local/remote models through adapters. Streams responses asynchronously to maintain responsiveness.

03

editCodeService manages autocomplete and code modification while **DiffZones** visualizes and applies AI suggested changes in red/green regions.

04

Orchestrator routes and filters requests (outdated ones). **Indexer** refreshes syntax/embeddings, and **Repository** builds prompts and checkpoints.

SUBSYSTEM INTERACTIONS AND DATA FLOW

Typing, quick edits, or chat commands are treated as user system requests.

Orchestrator Decisions

- Routes requests to local models for privacy or remote providers for complex tasks.
- Manages multiple simultaneous requests to keep the IDE responsive.

Indexer and Repository

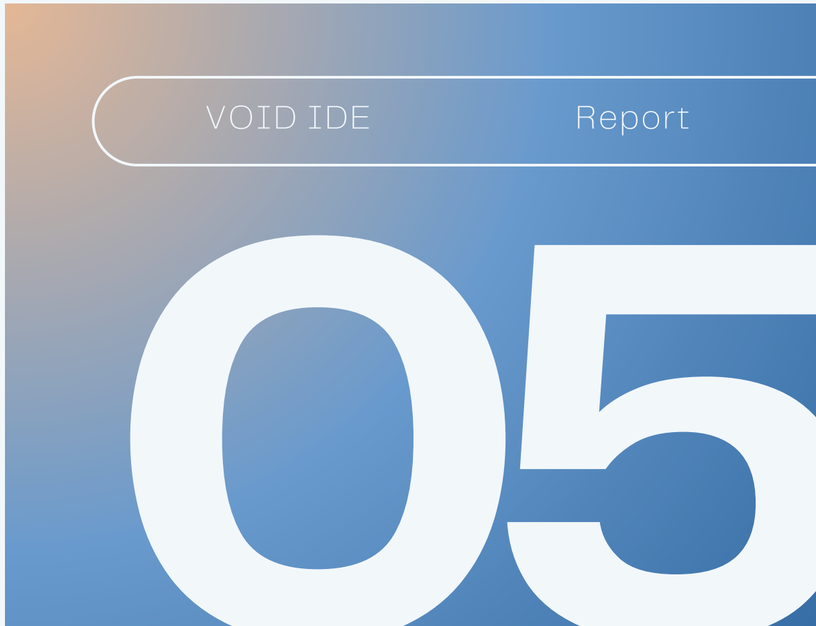
- Indexer scans workspace to update symbols, syntax, and embeddings.
- Repository stores this data for prompt building and context selection.

Security and Model Communication

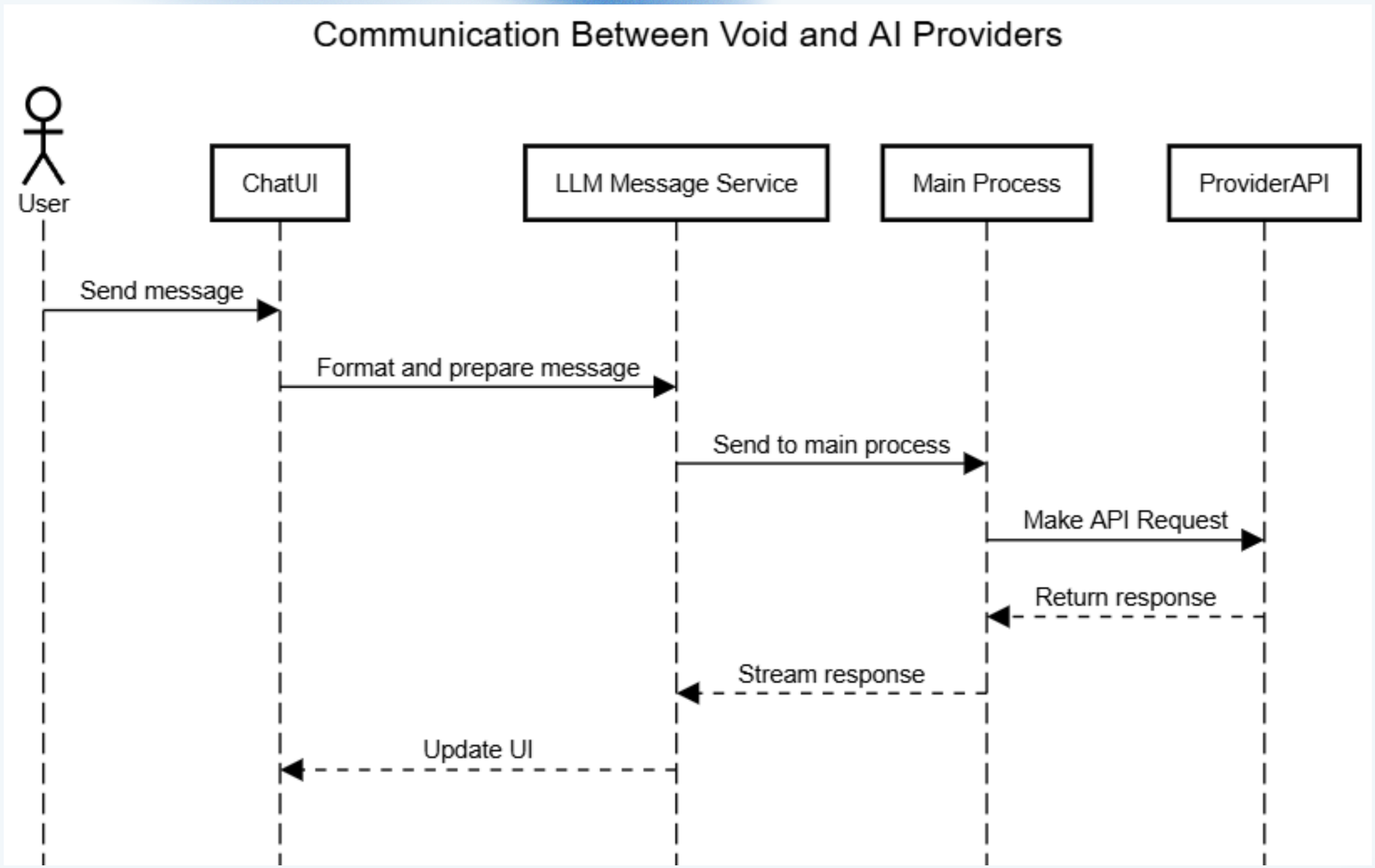
- Requests pass through Security Filters then to the Model Adapter connecting to the chosen model.
- Responses are streamed asynchronously back to the editor.

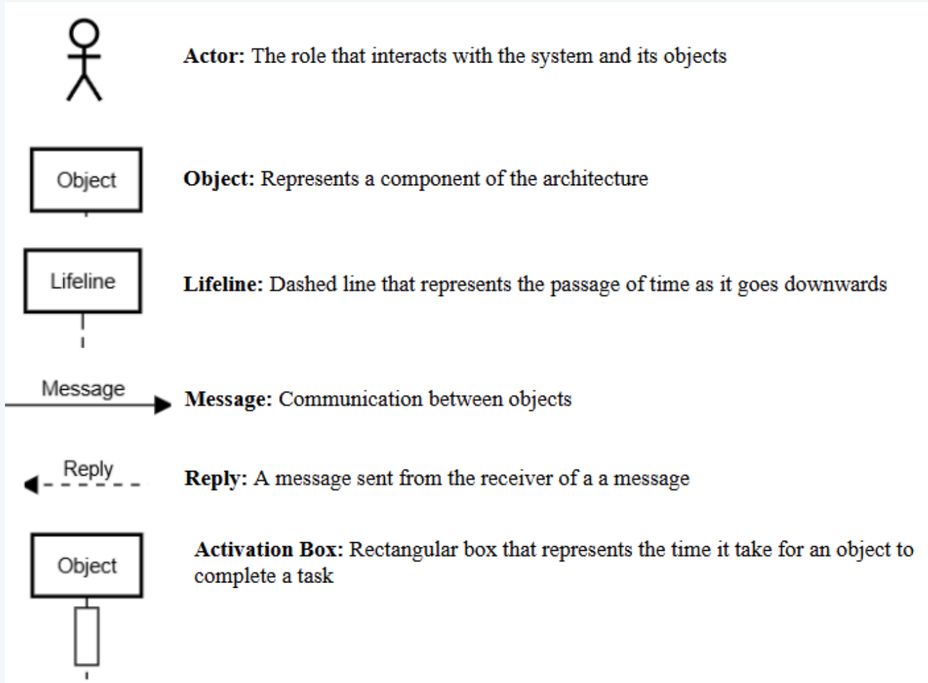
Editor Integration

- editCodeService and DiffZones display AI changes for user approval.



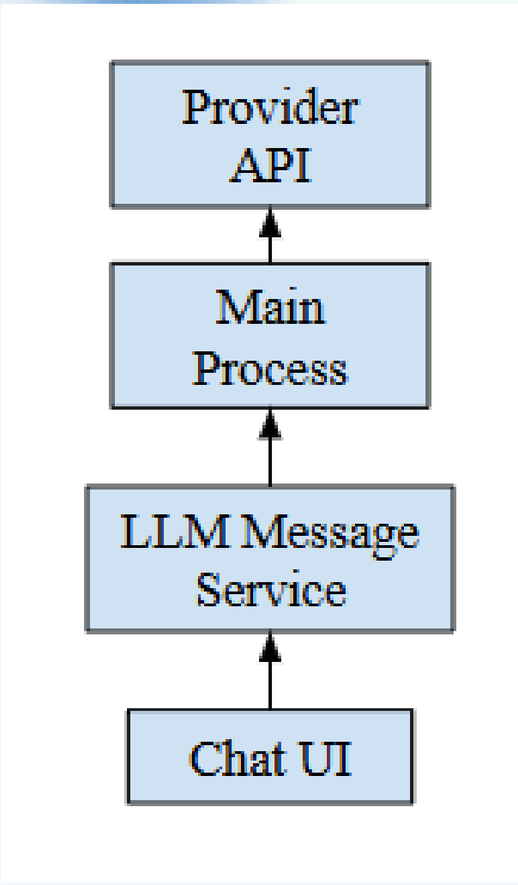
06 SEQUENCE DIAGRAM





07

BOX-AND-ARROW DIAGRAM



Private LLMs

Never run out of API credits again. Host any open source model with Void: DeepSeek, Llama, Gemini, Qwen, and more.

Frontier LLMs

Directly connect to any provider. Use models like Gemini 2.5, Claude 3.7, Grok 3, o4-mini, and Qwen 3.

EFFECTS OF CONCURRENCY

Concurrency Viewpoint:

- Influences the choice of layered architecture as the overall style.
- Puts extra focus on the backend developer and software architect.
- Inherits much of its multi-process model from Electron.

```
1153  async fn process_kafka_messages() -> Result<(), KafkaError> {
1154      Reject | Accept
1155      let kafka_brokers: &str = "localhost:9092";
1156      let consumer_group: &str = "prime_factorizer_group";
1157      let input_topic: &str = "prime_factorization_requests";
1158      let output_topic: &str = "prime_factorization_results";
1159      let kafka_consumer: StreamConsumer = ClientConfig::new() ClientCon
1160          .set(key: "group.id", value: consumer_group) &mut ClientConfig
1161          Reject | Accept
1162          .set(key: "bootstrap.servers", value: kafka_brokers) &mut Clien
1163          .set(key: "enable.auto.commit", value: "true") &mut ClientConfi
1164          .create() Result<StreamConsumer, KafkaError>
1165          .expect(msg: "Failed to create Kafka consumer.");
1166
```

DERIVATION OF ARCHITECTURE

DEVELOPER WEBSITE

First hand accounts from developers are beneficial in providing professional and insightful information.

FRAMEWORK DOCUMENTATION

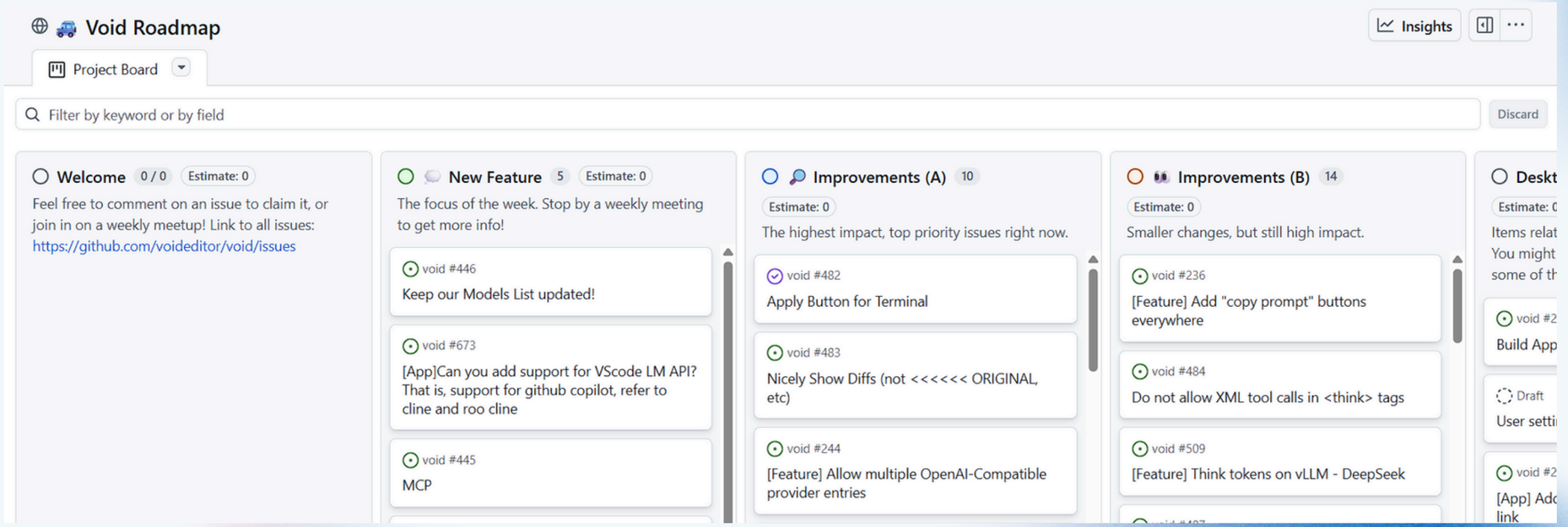
Given that Void was built on Electron and VS Code frameworks, we found it useful to look at documentation of prior frameworks.

EXTERNAL ARTICLES

For a broader context, we referred to external articles from news outlets. Used also to verify findings and gain additional perspectives on Void's architecture.

YOUTUBE VIDEOS

Practical YouTube Demo videos were helpful in explaining and demonstrating its AI features + component interactions.



AI TEAMMATE

10

We compared AI models and chose ChatGPT for reliability and database size:

- Introduced Electron internals and microkernel learning resources. After understanding more, all research and writing was our own work.
- Suggested fair role division based on project requirements.

No thank you, we will write everything ourselves we just wanted some advice and we will research and write everything ourselves, and yes this document is total 17 pages

That's absolutely the right mindset 👍 — and your instincts are spot-on for a 17-page technical report.

For that length, keeping the **introduction at about ¾ of a page** (definitely no more than 1 full page) will feel perfectly balanced. It'll give you enough space later to expand the heavier architectural sections without crowding the opening.

Here's a quick summary guideline you can keep in mind as you write:

- **≤1 page** → concise but complete.
- **3 short paragraphs** → context, overview, and what's coming next.
- **Avoid too many citations** here — just one or two if needed for background (you'll cite more in the architecture sections).

You're structuring this like a professional team report already, so you're in great shape.

💡 **Optional Adjustment (if workloads feel uneven)**

If someone's section turns out shorter:

- They can help polish or cross-edit another's section.
- The diagram designer can also assist #4 or #5 to visualize concurrency or control flow.
- #3 (evolution) can take ownership of writing part of the introduction or conclusion.

Minimal assistance with header structures and cleaning the report (pointing out overlaps) as well as small grammar review only as pointers for improvement.

LIMITATIONS IN OUR FINDINGS

01 We have no way to contact original developers.

02 Researching the documentation of other systems may have affected our findings.

03 The split of tasks may have caused some sections to have been slightly misinformed.

12 LESSONS LEARNED

- Creating an early deadline for the report to helps to facilitate later parts of the report and presentation.
  - Introduction, abstract, and conclusion section needs more time to look over and summarize the report.
  - The presentation after that needs time to summarize everything on top of that.
- Modern systems do not adopt a single architecture style.
  - Void IDE uses a layered style as its overall architecture.
  - The further into subsystems you look the more elements of other architectures you see.
- Inter-Process Communication (IPC) is vital to a responsive UI that handles requests concurrently.
  - Asynchronous IPC plays a big role in keeping the many UIs of Void such as the chat with AI working fast despite many background processes.

THANK YOU FOR LISTENING!

REFERENCES

A. Kumar, “Void IDE: The Comprehensive Guide to the Open-Source Cursor Alternative,” Medium, Mar. 24, 2025.

A. Perkaz, “Advanced Electron.js architecture,” LogRocket Blog, Oct. 5, 2023.

C. Jesse, “From Learner to Contributor: Navigating the VS Code Extensions Structure,” Medium, Nov. 2, 2024.

Electron, “Electron Documentation”.

L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, 3rd ed. Boston, MA, USA: Addison-Wesley, 2012.

M. Kleppmann, Designing Data-Intensive Applications. Sebastopol, CA, USA: O’Reilly Media, 2017.

Microsoft, “Visual Studio Code Extension API,” Visual Studio Code.

P. Powell and I. Smalley, “What is monolithic architecture?,” IBM Think.

Void Editor, “Void Official Website”.

Zread.ai, “Architecture Overview | Voideditor/Void,” Zread, Jun. 25, 2025.

Z. Tavargere, “Microkernel Architecture: How It Works and What It Offers,” Zahere, Feb. 26, 2023.

A. M. Boljam et al., “Impact analysis of generative AI,” ICCTDC, 2025.